Microsoft® BASIC

BASIC Language Reference

Version 7.0
For IBM® Personal Computers
and Compatibles

Microsoft Corporation

Simultaneously published in the U.S. and Canada.

Printed and bound in the United States of America.

Document No. DB0118-700-R00-1089

Table of Contents

Part 2 Add-On-Library Reference

Add-On-Library Summary Tables 421

Part 3 BASIC Toolbox Reference

Toolbox Summary Tables 491

Matrix Math Toolbox 501

Presentation Graphics Toolbox 507

Fonts Toolbox 518

Tables

Figures

Introduction

The printed documentation for Microsoft® BASIC 7.0 Professional Development System consists of three books: *Getting Started*, the *BASIC Language Reference*, and the *Programmer's Guide*.

Getting Started contains detailed information on setting up and customizing Microsoft BASIC to optimize program size and speed. It also introduces new features of this version of Microsoft BASIC and provides a brief introduction to the new Microsoft Advisor online Help system.

The *BASIC Language Reference* has these parts:

- Part 1, "Language Reference," contains reference material and programming examples for each BASIC function, statement, and metacommand, organized alphabetically.
- Part 2, "Add-On-Library Reference," contains reference material on the **SUB** and **FUNCTION** procedures that are supported by the add-on libraries included with Microsoft BASIC. These add-on libraries give you date/time, financial, and format functions.
- Part 3, "BASIC Toolbox Reference," contains reference material on the **SUB** and **FUNCTION** procedures that are supported by the toolbox files included with Microsoft BASIC. The toolboxes are Matrix Math, Presentation Graphics, Font, and User Interface.
- The appendixes include a list of keyboard scan and ASCII character codes, a list of BASIC reserved words, a command-line tools quick reference, reference information on error messages, and international character sort order tables.

The *Programmer's Guide* contains topical information about programming concepts and techniques and command-line tools.

Document Conventions

This manual uses the following typographic conventions:

Example of convention	Description
BASIC.LIB, ADD.EXE, COPY, LINK, /X	Uppercase (capital) letters indicate filenames and DOS-level commands. Uppercase also is used for command-line options (unless the application accepts only lowercase).

SUB, IF, LOOP, WHILE, TIME$	Bold uppercase letters indicate language-specific keywords with special meaning to Microsoft BASIC. Keywords are a required part of statement syntax, unless they are enclosed in double brackets as explained later in this table. In programs you write, you must enter keywords exactly as shown. However, they are not case sensitive; you can use any combination of uppercase or lowercase leters.
DateSerial#, MatMult, DefaultChart	Bold words with initial capital letters indicate functions and statements in add-on libraries or **FUNCTION** and **SUB** procedures in the BASIC toolboxes.
`CALL NewProc(arg1!, var2%)`	This type is used for program examples, program output, error messages and words you type.

`$INCLUDE:'BC.BI'`
`.`
`.`
`.`
`CHAIN "PROG1"`
`END`
	A column of three dots in an example indicates that part of the example program has been intentionally omitted.

`' Make one pass.`	The apostrophe (single right quotation mark) marks the beginning of a comment in sample programs.
filespec$	Italic letters indicate placeholders for information you must supply, such as a filename. The last character of the placeholder often indicates the data type of the information you must supply. Italics are also occasionally used in the text for emphasis.
⟦*expressionlist*⟧	Items inside double square brackets are optional.
{**WHILE** | **UNTIL**}	Braces and a vertical bar indicate a mandatory choice among two or more items. You must choose one of the items unless all of the items also are enclosed in double square brackets.
⟦*directory*⟧...	Three dots following an item indicate that more items having the same form may appear.

Ctrl key	Initial capital letters are used for the names of keys and key sequences, such as Del and Ctrl+R. The key names used in this manual correspond to the names on the IBM Personal Computer keys. Other machines may use different names. The carriage-return key, sometimes marked with a bent arrow, is referred to as Enter.
Alt+F1	A plus (+) indicates a combination of keys. For example, Alt+F1 means to hold down the Alt key while pressing the F1 key.
Down Arrow key	The cursor-movement ("arrow") keys are called direction keys. Individual direction keys are referred to by the direction of the arrow on the key top (Left, Right, Up, or Down).
"defined term"	Quotation marks usually indicate a new term defined in the text.
Video Graphics Array (VGA)	Acronyms and abbreviations usually are spelled out the first time they are used.

The following syntax (for the **LOCK...UNLOCK** statement) illustrates many of the typographic conventions in this manual:

LOCK ⟦#⟧*filenumber%*⟦,{*record&*|⟦*start&*⟧ **TO** *end&*}⟧
.
.
.
UNLOCK ⟦#⟧*filenumber%*⟦,{*record&*|⟦*start&*⟧ **TO** *end&*}⟧

Note Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft® Operating System/2 (MS® OS/2) and IBM OS/2. Similarly, the term "DOS" refers to the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system. The term "BASICA" refers to interpreted versions of BASIC in general.

Reference Page Format

Each function and statement description in Part 1, "Language Reference," uses the following format. The format also is used for functions and statements in Part 2, "Add-On-Library Reference," and for the **SUB** and **FUNCTION** procedures in Part 3, "BASIC Toolbox Reference."

Heading	Function
Action	Summarizes what the function or statement does.
Syntax	Gives the correct syntax for the function or statement.
Remarks	Describes arguments and options in detail and explains how to use the function or statement.
BASICA	Optional section that tells whether the given function or statement is new, enhanced, or different from the same statement in the Microsoft BASIC 2.0 Interpreter described in the IBM Personal Computer BASICA reference manual.
Note	Calls attention to important information about using the function or statement (also an optional section).
Warning	Urges caution in using the function or statement in a way that could delete files, produce unpredictable results, or cause the operating system to fail (also an optional section).
See Also	Mentions related functions and statements (also an optional section).
Example	Gives sample commands, programs, and program segments that illustrate the use of the function or statement (also an optional section).

Programming Style in This Manual

The following guidelines were used in writing programs in this manual. These guidelines are only recommendations for program readability; you are not obliged to follow them when writing your own programs.

- Keywords and symbolic constants appear in uppercase letters:

```
' PRINT, DO, LOOP, UNTIL are keywords.
PRINT "Title Page"
DO LOOP UNTIL Response$ = "N"

' FALSE and TRUE are symbolic constants
' equal to 0 and -1, respectively.
CONST FALSE = 0, TRUE = NOT FALSE
```

- Variable names are lowercase with an initial capital letter; variable names with more than one syllable may contain other capital letters to clarify the division:

```
NumRecords% = 45
DateOfBirth$ = "11/24/54"
```

- Line labels are used instead of line numbers. The use of line labels is restricted to event-trapping and error-handling routines, as well as **DATA** statements when used with **RESTORE**:

```
' TimerHandler and ScreenTwoData are line labels.
ON TIMER GOSUB TimerHandler
RESTORE ScreenTwoData
```

- As noted in an earlier section, an apostrophe (') introduces comments:

```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Control-flow blocks and statements in **SUB** and **FUNCTION** procedures are indented from the enclosing code:

```
SUB GetInput STATIC
   FOR I% = 1 TO 10
      INPUT X
      IF X > 0 THEN
         .
         .
         .
      ELSE
         .
         .
         .
      END IF
   NEXT I%
END SUB
```

- Lines longer than 80 characters may be continued on the next line using a line-continuation character (_). This is done to fit the example on the printed page. The QBX environment supports an infinite line length, and therefore does not use the line-continuation character; however, it will rejoin continued lines automatically when they are loaded from outside the environment.

```
DECLARE SUB StringAssign(BYVAL SrcSeg, BYVAL SrcOffset,_
                         DestSeg, BYVAL DestOffset,_
                         BYVAL DestLen)
```

Part 1

Language Reference

Part 1 is a reference for each function, statement, and metacommand in the BASIC language. Each entry is listed alphabetically and describes the action performed, and the syntax to use. Arguments, options, and typical usage are explained further under "Remarks." The "See Also" sections refer you to related entries. In addition, example programs are included with the descriptions so you can see exactly how a function or statement works.

In the beginning of Part 1 are summary tables for selected BASIC functions and statements. Each table contains entries that perform related tasks. Within each table, you can quickly see specific tasks you can accomplish, the functions or statements to use, and the actions that result.

BASIC Functions and Statements Summary Tables

The sections in this chapter are made up of related BASIC functions and statements that have been grouped together according to programming tasks. Each group's functions and statements are presented in tabular form, listing the type of task performed by each function or statement (for example, looping or searching), and the corresponding program action that takes place.

The following topics are summarized:

- Control-flow functions and statements
- Procedure-related statements
- Standard I/O functions and statements
- File I/O functions and statements
- ISAM File I/O functions and statements
- String-processing functions and statements
- Graphics functions and statements
- Trapping functions and statements

You can use these tables both as a reference guide to what each statement or function does and as a way to identify related statements.

Action statements shown in italic indicate a function or statement that is new to BASIC Version 7.0

Note Not all BASIC functions and statements are listed in these tables, only those recommended for structured programming. For complete information on all functions and statements, see the alphabetic entries that follow these tables.

Control-Flow Functions and Statements

Table 1.1 lists the BASIC statements used to control the flow of a program's execution.

Table 1.1 Summary of Control-Flow Functions and Statements

Task	Function or statement	Action
Looping	**FOR...NEXT**	Repeats statements between **FOR** and **NEXT** a specific number of times.
	EXIT FOR	Provides an alternative way to exit a **FOR...NEXT** loop.
	DO...LOOP	Repeats statements between **DO** and **LOOP**, either until a given condition is true (**DO...LOOP UNTIL** *condition*), or while a given condition is true (**DO...LOOP WHILE** *condition*).
	EXIT DO	Provides an alternative way to exit a **DO...LOOP** loop.
	WHILE...WEND	Repeats statements between **WHILE** and **WEND** while a given condition is true (similar to **DO WHILE** *condition*...**LOOP**).
Making decisions	**IF...THEN...ELSE** **END IF**	Conditionally executes or branches to different statements.
	SELECT CASE **END SELECT**	Conditionally executes different statements.
Managing the stack	**STACK** Function	*Returns maximum stack size that can be allocated.*
	STACK Statement	*Sets stack to new size.*
Exiting the program	**END** *number* **SYSTEM** *number* **STOP** *number*	Exit the program and set error level defined by *number*. If *number* is not present, exit the program with error level set to 0.

Procedure-Related Statements

Table 1.2 lists the statements used to define, declare, call, and pass arguments to BASIC procedures, and statements used to share variables among procedures, modules, and separate programs.

Table 1.2 Summary of Procedure-Related Statements

Task	Statement	Action
Defining a procedure	**FUNCTION... END FUNCTION**	Marks the beginning and end, respectively, of a **FUNCTION** procedure.
	SUB...END SUB	Marks the beginning and end, respectively, of a **SUB** procedure.
Calling a procedure	**CALL**	Transfers control to a **BASIC SUB** procedure, or to a procedure written in another programming language and compiled separately. (The **CALL** keyword is optional if a **DECLARE** statement is used.)
Exiting from a procedure	**EXIT FUNCTION**	Provides an alternative way to exit a **FUNCTION** procedure.
	EXIT SUB	Provides an alternative way to exit a **SUB** procedure.
Declaring a reference to a procedure	**DECLARE**	Declares a **FUNCTION** or **SUB** procedure and, optionally, specifies the number and type of its parameters.
Sharing variables among modules, procedures, or programs	**COMMON**	Shares variables among separate modules. When used with the **SHARED** attribute, **COMMON** blocks share variables among different procedures in the same module. When used with a blank (unnamed) **COMMON** block, variables pass from current program to new program when control is transferred with the **CHAIN** statement.

Table 1.2 Continued

Task	Statement	Action
	SHARED	When used with the **COMMON**, **DIM**, or **REDIM** statement at the module level (for example, **DIM SHARED**), shares variables with every **SUB** or **FUNCTION** procedure in a single module. When used by itself within a procedure, shares variables between that procedure and the module-level code.
Preserving variable values	**STATIC**	Forces variables to be local to a procedure or **DEF FN** function and preserves the value stored in the variable if the procedure or function is exited, then called again.

Standard I/O Functions and Statements

Table 1.3 lists the functions and statements used in BASIC for standard I/O (typically, input from the keyboard and output to the screen).

Table 1.3 Summary of Standard I/O Functions and Statements

Task	Function or statement	Action
Printing text on the screen	**PRINT**	Outputs text to the screen. Using **PRINT** with no arguments creates a blank line.
	PRINT USING	Outputs formatted text to the screen.
Changing the width of the output line	**WIDTH**	Changes the width of the screen to either 40 columns or 80 columns. Also, on computers with an EGA or VGA, controls the number of lines on the screen (25 or 43).
	WIDTH "SCRN:"	Assigns a maximum length to lines output to the screen when used before an **OPEN "SCRN:"** statement.
Getting input from the keyboard	**INKEY$**	Reads a character from the keyboard (or a null string if no character is waiting).

Table 1.3 Continued

Task	Function or statement	Action
	INPUT$	Reads a specified number of characters from the keyboard and stores them in a single string variable.
	INPUT	Reads input from the keyboard and stores it in a list of variables.
	LINE INPUT	Reads a line of input from the keyboard and stores it in a single string variable.
Positioning the cursor on the screen	**LOCATE**	Moves the cursor to a specified row and column. Also sets cursor size and turns it on and off.
	SPC	Skips a specified number of spaces in printed output.
	TAB	Displays printed output in a given column.
Getting information on cursor location	**CSRLIN**	Tells which row or line position the cursor is in.
	POS(*n*)	Tells which column the cursor is in.
Creating a text viewport	**VIEW PRINT**	Sets the top and bottom rows for displaying text output.

File I/O Functions and Statements

Table 1.4 lists the functions and statements used in BASIC file operations.

Table 1.4 Summary of File I/O Functions and Statements

Task	Function or statement	Action
Creating a new file or accessing an existing file or device	**OPEN**	Opens a file or device for retrieving or storing records (I/O).
Closing a file or device	**CLOSE**	Ends I/O to a file or device.
Storing data in a file	**PRINT #**	Stores a list of variables as record fields in a previously opened file.[1]
	PRINT # USING	Similar to **PRINT #**, except **PRINT # USING** formats the record fields.[1]

Table 1.4 *Continued*

Task	Function or statement	Action
	WRITE #	Stores a list of variables as record fields in a previously opened file.[1]
	WIDTH #	Specifies a standard width for each record in a file.[1]
	PUT	Stores the contents of a user-defined variable in a previously opened file.[2]
Retrieving data from a file	**INPUT #**	Reads fields from a record and assigns each field in the record to a program variable.[1]
	INPUT$	Reads a string of characters from a file.
	LINE INPUT #	Reads a record and stores it in a single string variable.[1]
	GET	Reads data from a file and assigns the data to elements of a user-defined variable.[2]
Getting information about a file	**EOF**	Tests whether all data has been read from a file.
	FILEATTR	Returns the number assigned by the operating system to an open file and a number that indicates the mode in which the file was opened (input, output, append, binary, random, or ISAM).
	LOC	Gives the current position within a file. With binary access, this is the byte position. With sequential access, this is the byte position divided by 128. With random access, this is the record number of the last record read or written.
	LOF	Gives number of bytes in open file.
	SEEK Function	Gives the location where the next I/O operation will take place. With random access, this is the number of the next record to be read or written. With all other kinds of file access, this is the byte position of the next byte to be read or written.

Table 1.4 Continued

Task	Function or statement	Action
Moving around in a file	**SEEK** Statement	Sets the byte position or record number for the next read or write operation in an open file.
Managing disk drives, directories, and files	**CHDRIVE**	*Changes the current drive.*
	CURDIR$	*Gives current directory specification for specified drive (or current drive if no drive is specified).*
	DIR$	On first call, returns the first file that matches the file specification. On successive calls (with no argument), returns remaining files matching the file specification.
	FILES	Prints a listing of the files in a specified directory.
	FREEFILE	Returns next available file number.
	KILL	Deletes a file from the disk.
	NAME	Changes a file's name.

1 For use with sequential files.
2 For use with binary or random-access files.

ISAM File I/O Functions and Statements

Table 1.5 lists the functions and statements used in BASIC ISAM programming.

Table 1.5 Summary of ISAM File I/O Functions and Statements

Task	Function or statement	Action
Opening and closing tables and databases	**OPEN** *database$* **FOR ISAM** *tabletype tablename$* **AS** [[#]] *filenumber%*	*Opens a database or creates a new database, opens a table or creates a new table.*
	CLOSE	*Closes a specified table.*
	TYPE *tabletype* *columnname* **AS** *tablename* {*columnname* **AS** *typename*} **END TYPE**	*Declares the type of the record that is used to build the table.*
Managing the data dictionary	**CREATEINDEX**	*Creates an index.*
	DELETEINDEX	*Deletes an index.*

Table 1.5 *Continued*

Task	Function or statement	Action
Positioning	**MOVEFIRST**	*Makes the first record current.*
	MOVELAST	*Makes the last record current.*
	MOVENEXT	*Makes the next record current.*
	MOVEPREVIOUS	*Makes the previous record current.*
	SEEKEQ **SEEKGE** **SEEKGT**	*Makes matching record current.*
	SETINDEX	*Makes the named index the current index.*
Getting information about a table	**BOF**	*Returns TRUE when the current position is beyond the first record of the current index.*
	EOF	*Returns TRUE when the current position is beyond the last record of the current index.*
	FILEATTR	*Returns information about an open ISAM table.*
	GETINDEX$	*Returns name of the current index.*
	LOF	*Returns the number of records in a table.*
Exchanging data	**INSERT**	*Inserts a record in a table.*
	RETRIEVE	*Retrieves the current record.*
	UPDATE	*Overwrites current record.*
	DELETE	*Deletes the current record.*
Transaction processing	**BEGINTRANS**	*Marks the beginning of a transaction.*
	CHECKPOINT	*Saves all currently open databases to disk.*
	COMMITTRANS	*Commits all changes since the most recent* **BEGINTRANS***.*
	ROLLBACK	*Rolls back data to its state at the savepoint.*
	ROLLBACK ALL	*Rolls back state of database to beginning of current transaction.*
	SAVEPOINT%	*Defines a savepoint and returns savepoint's reference number.*

Table 1.5 *Continued*

Task	Function or statement	Action
Comparing text	**TEXTCOMP**	*Compares two strings as they would be compared by ISAM.*

String-Processing Functions and Statements

Table 1.6 lists the functions and statements available in BASIC for working with strings.

Table 1.6 Summary of String-Processing Functions and Statements

Task	Function or statement	Action
Getting part of a string	**LEFT$**	Returns given number of characters from the left side of a string.
	RIGHT$	Returns given number of characters from the right side of a string.
	LTRIM$	Returns a copy of a string with leading spaces stripped away.
	RTRIM$	Returns a copy of a string with trailing spaces stripped away.
	MID$ function	Returns given number of characters from anywhere in a string.
Calculating available memory space	**FRE**(*a$*)	*For far strings, returns the space remaining in* a$*'s segment. For near strings, compacts string storage space and returns space remaining in DGROUP.*
	FRE(*literal string*)	*For far strings, returns the space remaining in the temporary string segment. For near strings, compacts string storage space and returns space remaining in DGROUP.*
	FRE(*number*)	Measures available storage for strings, nonstring arrays, or the stack.
Searching strings	**INSTR**	Searches for a string within another string.
Converting to uppercase or lowercase letters	**LCASE$**	Returns a copy of a string with all uppercase letters (A–Z) converted to lowercase letters (a–z); leaves lowercase letters and other characters unchanged.

Table 1.6 *Continued*

Task	Function or statement	Action
	UCASE$	Returns a copy of a string with all lowercase letters (a–z) converted to uppercase letters (A–Z); leaves uppercase letters and other characters unchanged.
Changing strings	**MID$** Statement	Replaces part of a string with another string.
	LSET	Left justifies a string within a fixed-length string.
	RSET	Right justifies a string within a fixed-length string.
Converting between numbers and strings	**STR$**	Returns the string representation of the value of a numeric expression.
	VAL	Returns the numeric value of a string expression.
Creating strings of repeating characters	**SPACE$**	Returns a string of blank characters of a specified length.
	STRING$	Returns a string consisting of one repeated character.
Getting the length of a string	**LEN**	Tells how many characters are in a string.
Working with ASCII values	**ASC**	Returns the ASCII value of the given character.
	CHR$	Returns the character with the given ASCII value.
Creating string pointers	**SSEG**	*Returns the segment of the string data.*
	SADD	Returns the offset of the string data.
	SSEGADD	*Returns a far pointer to the string data.*
	VARPTR	Returns the offset address of a variable or string descriptor.
	VARSEG	Returns the segment address of a variable or string descriptor.

Graphics Functions and Statements

Table 1.7 lists the functions and statements used in BASIC for pixel-based graphics.

Table 1.7 Summary of Graphics Functions and Statements

Task	Function or statement	Action
Setting screen-display characteristics	**SCREEN** Statement	Specifies a BASIC screen mode, which determines screen characteristics such as graphics capability, resolution, and color-number range.
	SCREEN Function	Returns a character's ASCII value or its color from a specified screen location.
Plotting or erasing a single point	**PSET**	Draws a pixel on the screen in the specified color, using the screen's foreground color as default.
	PRESET	Draws a pixel on the screen in the specified color, using the screen's background color as default.
Drawing shapes	**LINE**	Draws a straight line or a box.
	CIRCLE	Draws a circle, ellipse, arc, or pie.
	DRAW	Draws an object. Combines many of the features of other BASIC graphics statements into a graphics macro language.
Defining screen coordinates	**VIEW**	Specifies a rectangle on the screen (or viewport) as the area for graphics output.
	WINDOW	Defines the dimensions of the current viewport.
	PMAP	Maps view coordinates to window coordinates, or vice versa.
	POINT(*number*)	Returns the screen coordinates of the graphics cursor, or the color of a specified pixel.
Using color	**COLOR**	Sets the default colors used in graphics output.
	PALETTE	Assigns different attributes to color numbers. Works only on systems equipped with an EGA or VGA.

Table 1.7 *Continued*

Task	Function or statement	Action
	POINT(*x*, *y*)	Returns the color number of a pixel whose screen coordinates are *x* and *y*.
Painting enclosed shapes	**PAINT**	Fills an area on the screen with a color or pattern.
Animating	**GET**	Copies a rectangular area on the screen by translating the image to numeric data and storing the data in a numeric array.
	PUT	Displays an image on the screen that was previously copied with **GET**.
	PCOPY	Copies one screen page to another.

Trapping Functions and Statements

Table 1.8 lists the functions and statements used by BASIC to trap and process errors and events.

Table 1.8 Summary of Trapping Functions and Statements

Task	Function or statement	Action
Trapping errors while a program is running	**ON ERROR GOTO** *line*	At the module level, causes a program to branch to *line*, where *line* refers to either a line number or line label. Branching takes place whenever an error occurs during execution.
	ON LOCAL ERROR GOTO *line*	*At the procedure level, causes a program to branch to* line, *where* line *refers to either a line number or line label. Branching takes place whenever an error occurs during execution of the procedure.*
	ON ERROR RESUME NEXT	*At the module level, causes an implicit error trap, then* ***RESUME NEXT****. Returns the error number in* ***ERR****.*
	ON LOCAL ERROR RESUME NEXT	*At the procedure level, causes an implicit error trap, then* ***RESUME NEXT****. Returns the error number in* ***ERR****.*

Table 1.8 Continued

Task	Function or statement	Action
	RESUME	Returns control to the program after executing an error-handling routine. The program resumes at either the statement causing the error (**RESUME**), the statement after the one causing the error (**RESUME NEXT**), or the line identified by *line* (**RESUME** *line*)
Getting error-status data	**ERR** Function	Returns the code for an error that occurs at run time.
	ERR Statement	*Sets* **ERR** *to an integer.*
	ERL	Returns the number of the line on which an error occurred (if program has line numbers).
	ERDEV	Returns a device-specific error code for the last device (such as a printer) for which the system detected an error.
	ERDEV$	Returns the name of the last device for which the system detected an error.
Defining your own error codes	**ERROR**	Simulates the occurrence of a BASIC error; can also be used to define an error not trapped by BASIC.
Trapping events while a program is running	**ON** *event* **GOSUB** *line*	Causes a branch to the subroutine starting with *line*, where *line* refers either to a line number or line label, whenever *event* occurs during execution.
	event **ON**	Enables trapping of *event.*
	event **OFF**	Disables trapping of *event.*
	event **STOP**	Suspends trapping of *event.*
	EVENT ON	*Enables event trapping.*

Table 1.8 *Continued*

Task	Function or statement	Action
	EVENT OFF	*Disables event trapping.*
	RETURN	Returns control to the program after executing an event-handling subroutine. The program resumes at either the statement immediately following the statement that called the subroutine (**RETURN**), or the line that is identified by *line* (**RETURN** *line*).

ABS Function

Action Returns the absolute value of a numeric expression.

Syntax **ABS**(*numeric-expression#*)

Remarks The absolute value is the unsigned magnitude of its argument. For example, `ABS(-1)` and `ABS(1)` are both 1.

Example The following example finds an approximate value for a cube root. It uses **ABS** to find the difference between two guesses to see if the current guess is accurate. **DEFDBL** establishes the default data type for all variables.

```
DEFDBL A-Z
Precision = .0000001#
CLS                                  ' Clear the screen.
INPUT "Enter a value: ", Value       ' Prompt for input.
' Make the first two guesses.
X1 = 0#: X2 = Value
' Loop until the difference between guesses is
' less than the required precision.
DO UNTIL ABS(X1 - X2) < Precision
    X = (X1 + X2) / 2#
    ' Adjust the guesses.
    IF X * X * X - Value < 0# THEN
        X1 = X
    ELSE
        X2 = X
    END IF
LOOP
PRINT "The cube root is "; X
```

Output

```
Enter a value: 27
The cube root is  2.99999997206032
```

Absolute Routine

Action Transfers control to a machine-language procedure.

Syntax **CALL Absolute** ([[*argumentlist*,]] *integervariable%*)

Remarks The **Absolute** routine uses the following arguments:

Argument	Description
argumentlist	Arguments passed to a machine-language procedure as offsets (near pointers) from the current data segment. Although arguments are passed as offsets, the machine-language program is invoked with a far call.
integervariable%	The offset from the current code segment, set by **DEF SEG**, to the starting location of the procedure. The argument *integervariable%* is not passed to the procedure. Your program may need to execute a **DEF SEG** statement before executing **Absolute** to set the code segment for the called routine. Using a noninteger value for *integervariable%* produces unpredictable results.

Note The **Absolute** routine is provided to maintain compatibility with earlier versions of BASIC. Mixed-language programming using the **CALL** and **DECLARE** statements provides a simpler way to use assembly language with BASIC.

When using the **Absolute** routine in OS/2 protected mode, be careful not to refer to an illegal memory address. Before it executes the **Absolute** routine, BASIC attempts to get an executable-code-segment alias for the code you wish to access. If this operation fails, BASIC generates the error message `Permission denied.` A safe place to store user-written machine code is in memory allocated for a conventional BASIC object, such as an array.

To use the **Absolute** routine in the QBX environment, use the QBX.QLB Quick library. To use the **Absolute** routine outside of the QBX environment, link your program with the QBX.LIB file.

The QBX.BI header file contains the necessary declarations for the **Absolute** routine.

BASICA Assembly language programs that are invoked from BASICA and that have string arguments must be changed, because string descriptors are now 4 bytes long. For a near-string descriptor, the 4 bytes are the low byte and high byte of the string length, followed by the low byte and high byte of the string address. Far-string descriptors also are 4 bytes long, but their structure is proprietary. For more information on using far-string descriptors, see Chapter 12, "Mixed-Language Programming" and Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

See Also **CALL, CALLS** Statements (Non-BASIC)

Example The following example uses **Absolute** to execute a machine-language program stored in an array. The program indicates whether a math coprocessor is installed.

```
CONST nASMBYTES = 14
DEFINT A-Z
DIM AsmProg(1 TO (nASMBYTES / 2))

' The machine-language program stored as data to read into the array.
AsmBytes:
DATA &H55                 : ' PUSH BP         Save base pointer.
DATA &H8B, &HEC           : ' MOV  BP,SP      Get our own.
DATA &HCD, &H11           : ' INT  11H        Make the ROM-BIOS call.
DATA &H8B, &H5E, &H06     : ' MOV  BX,[BP+6]  Get argument address.
DATA &H89, &H07           : ' MOV  [BX],AX    Save list in argument.
DATA &H5D                 : ' POP  BP         Restore base pointer.
DATA &HCA, &H02, &H00     : ' RET  2          Pop argument off stack
                            '                 and make far return.

' Poke the machine-language program into the array.
P = VARPTR(AsmProg(1))        ' Get the starting offset of the array.
DEF SEG = VARSEG(AsmProg(1))  ' Change the segment.
FOR I = 0 TO nASMBYTES - 1
   READ J
   POKE (P + I), J
NEXT I

' Execute the program. The program expects a single integer argument.
CALL Absolute(X%, VARPTR(AsmProg(1)))
DEF SEG   ' Restore the segment.
' X% now contains bit-encoded equipment list returned by the system.
' Mask off all but the coprocessor bit (bit 2).
CoProcessor = X% AND &H2
IF CoProcessor = 2 THEN
   PRINT "Math coprocessor present."
ELSE
   PRINT "No math coprocessor."
END IF
```

ASC Function

Action Returns a numeric value that is the ASCII code for the first character in a string expression.

Syntax **ASC**(*stringexpression$*)

Remarks If the argument *stringexpression$* is null, BASIC generates the run-time error message `Illegal function call.`

See Also **CHR$**; Appendix A, "Keyboard Scan Codes and ASCII Character Codes"

Example The following example uses **ASC** to calculate a hash value—an index value for a table or file—from a string:

```
CONST HASHTABSIZE = 101
CLS                                ' Clear the screen.
INPUT "Enter a name: ",Nm$         ' Prompt for input.
TmpVal = 0
FOR I=1 TO LEN(Nm$)
    ' Convert the string to a number by summing the values
    ' of individual letters.
    TmpVal = TmpVal + ASC(MID$(Nm$,I,1))
NEXT I
    ' Divide the sum by the size of the table.
    HashValue = TmpVal MOD HASHTABSIZE
PRINT "The hash value is "; HashValue
```

Output

```
Enter a name: Bafflegab
The hash value is  66
```

ATN Function

Action Returns the arctangent of a numeric expression.

Syntax **ATN**(*numeric-expression*)

Remarks The argument *numeric-expression* can be of any numeric type.

ATN (arctangent) is the inverse of **TAN** (tangent). **ATN** takes the ratio of two sides of a right triangle (*numeric-expression*) and returns the corresponding angle. The ratio is the ratio of the lengths of the side opposite the angle and the side adjacent to the angle.

The result is given in radians and is in the range $-\pi/2$ to $\pi/2$ radians, where $\pi = 3.141593$ and $\pi/2$ radians = 90 degrees. **ATN** is calculated in single precision if *numeric-expression* is an integer or single-precision value. If you use any other numeric data type, **ATN** is calculated in double precision.

To convert values from degrees to radians, multiply the angle (in degrees) times $\pi/180$ (or .0174532925199433). To convert a radian value to degrees, multiply it by $180/\pi$ (or 57.2957795130824). In both cases, $\pi \approx 3.141593$.

See Also **COS, SIN, TAN**

Example The following example first finds the tangent of `PI/4` and then takes the arctangent of the value. The result is `PI/4`.

```
CONST PI=3.141592653
PRINT ATN(TAN(PI/4.0)), PI/4.0
```

Output

```
78539816325   .78539816325
```

BEEP Statement

Action Makes a sound through the speaker.

Syntax **BEEP**

Remarks The **BEEP** statement makes a sound through the loudspeaker. This statement makes the same sound as the following statement:

```
PRINT CHR$(7)
```

Example The following example uses **BEEP** to indicate an error in the response:

```
CLS                  ' Clear the screen.
DO
   INPUT "Hear a beep (Y or N)"; Response$
   R$ = UCASE$ (MID$ (Response$,1,1))
   IF R$ <> "Y" OR R$ = "N" THEN EXIT DO
   BEEP
LOOP
```

BEGINTRANS Statement

Action Indicates the beginning of a transaction (a series of ISAM database operations).

Syntax **BEGINTRANS**

Remarks Transactions are a way to group a series of ISAM operations so that you can commit them as a whole, rescind them all, or rescind operations since a designated savepoint. Use the **COMMITTRANS** statement to commit a transaction. Use the **SAVEPOINT** function to designate a savepoint. Use the **ROLLBACK** and **ROLLBACK ALL** statements to rescind all or part of a transaction's operations.

If you attempt to use **BEGINTRANS** when there already is a transaction pending, BASIC generates the error message `Illegal function call.`

Any ISAM operation that closes a table causes transactions to be committed. For example, if a type mismatch occurs while you are opening an ISAM table, the table is closed and a pending transaction is committed.

You may wish to code your programs so they open all tables, then perform all transactions, then close tables. Make sure any operation that can close a table occurs outside a transaction.

See Also **COMMITTRANS, ROLLBACK, SAVEPOINT**

Example The following example uses the **DELETE** statement to remove records from an ISAM file. It creates a transaction with the **BEGINTRANS** and **COMMITTRANS** statements, and uses **SAVEPOINT** and **ROLLBACK** to provide rollback of any or all of the deletions.

The program uses the file called BOOKS.MDB, which SETUP copies to your disk.

```
DEFINT A-Z
TYPE Borrower
   Cardnum AS LONG              ' Card number.
   TheName AS STRING * 36       ' Name.
   Address AS STRING * 50       ' Address.
   City AS STRING * 26          ' City.
   State AS STRING * 2          ' State.
   Zip AS LONG                  ' Zip code.
END TYPE

DIM People AS Borrower          ' Record structure variable.
CONST Database = "BOOKS.MDB"    ' Name of the disk file.
CONST Tablename = "Cardholders" ' Name of the table.
CONST viewbottom = 17, viewtop = 3, msgtxt = " *** Deleted: Savepoint "
DIM SavePts(viewbottom - viewtop + 1)
TableNum = FREEFILE
OPEN Database FOR ISAM Borrower Tablename AS TableNum
```

```
' Loop until user chooses to quit.
DO
    VIEW PRINT
    CLS : COLOR 15, 0
    PRINT SPC(34); "Card Holders"
    PRINT "Card#"; SPC(2); "Name"; SPC(13); "Address";
    PRINT SPC(20); "City"; SPC(6); "State"; SPC(2); "Zip"
    LOCATE 24, 1: PRINT "Choose a key:    ";
    PRINT "D - Delete this record    N - Next record    Q - Quit";
    MOVEFIRST TableNum
    VIEW PRINT viewtop TO viewbottom: COLOR 7, 0: CLS
    vPos = viewtop

    ' Loop through a screenful of records.
    DO
        ' For each record, display and ask user what to do with it.
        RETRIEVE TableNum, People
        LOCATE vPos, 1
        PRINT USING ("#####"); People.Cardnum;
        PRINT "  "; LEFT$(People.TheName, 15); "  ";
        PRINT LEFT$(People.Address, 25); "  ";
        PRINT LEFT$(People.City, 10); "  "; People.State; "   ";
        PRINT USING ("#####"); People.Zip

        ' Get keystroke from user.
        validkeys$ = "DNQ"
        DO
            keychoice$ = UCASE$(INKEY$)
        LOOP WHILE INSTR(validkeys$, keychoice$) = 0 OR keychoice$ = ""

        ' Process keystroke.
        SELECT CASE keychoice$
        CASE "D"
            IF NOT inTransaction THEN
                inTransaction = -1
                BEGINTRANS
            END IF
            NumSavePts = NumSavePts + 1
            SavePts(NumSavePts) = SAVEPOINT
            DELETE TableNum
            LOCATE vPos, 7: PRINT msgtxt; NumSavePts; SPC(79 - POS(0));
```

```
        CASE "Q"
            EXIT DO
        CASE "N"
            MOVENEXT TableNum
        END SELECT
        vPos = vPos + 1
    LOOP UNTIL EOF(TableNum) OR vPos = viewbottom

    ' If user didn't delete any records, simply quit.
    IF NumSavePts = 0 THEN EXIT DO

    ' Allow user to commit deletions, or roll back some or all of them.
    VIEW PRINT: LOCATE 24, 1: PRINT "Choose a key:    ";
    PRINT "R - Rollback to a savepoint   A - Rollback all deletions"
    PRINT SPC(17); "Q - commit deletions and Quit";
    validkeys$ = "RAQ"
    DO
        keychoice$ = UCASE$(INKEY$)
    LOOP WHILE INSTR(validkeys$, keychoice$) = 0 OR keychoice$ = ""
    SELECT CASE keychoice$
    CASE "R"
        VIEW PRINT 24 TO 25: PRINT : PRINT
        DO
            PRINT "Roll back to which savepoint ( 1 -"; NumSavePts; ")";
            INPUT RollbackPt
        LOOP UNTIL RollbackPt > 0 AND RollbackPt <= NumSavePts
        ROLLBACK SavePts(RollbackPt)
        NumSavePts = RollbackPt - 1
    CASE "A"
        NumSavePts = 0
        ROLLBACK ALL
    CASE "Q"
        EXIT DO
    END SELECT
LOOP

IF inTransaction THEN COMMITTRANS
CLOSE
END
```

BLOAD Statement

Action Loads a memory-image file created by **BSAVE** into memory from an input file or device.

Syntax BLOAD *filespec$*⟦,*offset%*⟧

Remarks The **BLOAD** statement allows a program or data saved as a memory-image file to be loaded anywhere in memory. A memory-image file is a byte-for-byte copy of what was originally in memory.

The **BLOAD** statement uses the following arguments:

Argument	Description
filespec$	A string expression that specifies the file or device from which to load a memory-image file.
offset%	The offset of the address where loading is to start.

The starting address for loading is determined by the specified offset and the most recent **DEF SEG** statement. If *offset%* is omitted, the segment address and offset contained in the file (the address used in the **BSAVE** statement) are used. Thus, the file is loaded at the address used when saving the file.

If you supply an offset, the segment address used is the segment set by the most recently executed **DEF SEG** statement. If there has been no **DEF SEG** statement, the BASIC data segment (DGROUP) is used as the default.

If the offset is a long integer, or a single-precision or double-precision number, it is converted to an integer. If the offset is a negative number between –1 and –32,768, inclusive, it is treated as an unsigned 2-byte offset.

Note Programs written in earlier versions of BASIC no longer work if they use **VARPTR** to access numeric arrays.

Because **BLOAD** does not perform an address-range check, it is possible to load a file anywhere in memory. You must be careful not to write over BASIC or the operating system.

Because different screen modes use memory differently, do not load graphic images in a screen mode other than the one used when they were created.

Because BASIC program code and data items are not stored in the same locations as they were in BASICA, do not use **BLOAD** with files created by BASICA programs.

BLOAD and Expanded Memory Arrays

Do not use **BLOAD** to load a file into an expanded memory array. If you start QBX with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays less than 16K in size.
- Fixed-length string arrays less than 16K in size.
- User-defined-type arrays less than 16K in size.

If you want to use **BLOAD** to load a file into an array, first start QBX without the /Ea switch. (Without the /Ea switch, no arrays are stored in expanded memory.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started.*

BASICA **BLOAD** does not support the cassette device.

See Also **BSAVE**; **DEF SEG**; **SSEG**; **VARPTR**, **VARSEG**

Example The following example uses **BLOAD** to retrieve and display a drawing of a magenta cube inside a box. The **BSAVE** statement programming example shows how the drawing was created and saved on disk in a file named MAGCUBE.GRH.

You must create the file MAGCUBE.GRH using the **BSAVE** statement programming example before you can run this program.

```
DIM Cube(1 TO 675)
' Set the screen mode. The mode should be the same as the
' mode used to create the original drawing.
SCREEN 1
' Set segment to the array Cube's segment and load
' the graphic file into Cube.
DEF SEG = VARSEG(Cube(1))
BLOAD "MAGCUBE.GRH", VARPTR(Cube(1))
DEF SEG                 ' Restore default BASIC segment.
' Put the drawing on the screen.
PUT (80, 10), Cube
```

BOF Function

Action Tests whether the current position is at the beginning of an ISAM table.

Syntax BOF(*filenumber%*)

Remarks **BOF** returns true (nonzero) if the current position is at the beginning of the table. The beginning of an ISAM table is the position before the first record according to the current index. The argument *filenumber%* is the number used in the **OPEN** statement to open the table.

See Also **EOF, LOC, LOF, OPEN**

Example See the **CREATEINDEX** statement programming example, which uses the **BOF** function.

BSAVE Statement

Action Transfers the contents of an area of memory to an output file or device.

Syntax **BSAVE** *filespec$*, *offset%*, *length%*

Remarks The **BSAVE** statement uses the following arguments:

Argument	Description
filespec$	A string expression that specifies the name of the file or device on which to save a memory-image file.
offset%	The offset of the starting address of the area in memory to be saved.
length%	The number of bytes to save. This is a numeric expression that returns an unsigned integer between 0 and 65,535, inclusive.

The **BSAVE** statement allows data or programs to be saved as memory-image files on disk. A memory-image file is a byte-for-byte copy of what is in memory along with control information used by **BLOAD** to load the file.

The starting address of the area saved is determined by the offset and the most recent **DEF SEG** statement.

Note Programs written in earlier versions of BASIC no longer work if they use **VARPTR** to access numeric arrays.

Because different screen modes use memory differently, do not load graphic images in a screen mode other than the one used when they were created.

If no **DEF SEG** statement is executed before the **BSAVE** statement, the program uses the default BASIC data segment (DGROUP). Otherwise, **BSAVE** begins saving at the address specified by the offset and by the segment set in the most recent **DEF SEG** statement.

If the offset is a long integer, or single- or double-precision floating-point value, it is converted to an integer. If the offset is a negative number between –1 and –32,768, inclusive, it is treated as an unsigned 2-byte offset.

BSAVE and Expanded Memory Arrays

Do not use **BSAVE** to transfer an expanded memory array to an output file. (If you start QBX with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays less than 16K in size.
- Fixed-length string arrays less than 16K in size.
- User-defined-type arrays less than 16K in size.

If you want to use **BSAVE** to transfer an array to an output file, first start QBX without the /Ea switch. (Without the /Ea switch, no arrays are stored in expanded memory.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started.*

BASICA **BSAVE** does not support the cassette device.

See Also **BLOAD, DEF SEG**

Example The following example draws a magenta cube inside a white box and then uses **BSAVE** to store the drawing in the file MAGCUBE.GRH. The **BLOAD** statement programming example shows how you can retrieve and display the drawing after you create it.

```
DIM Cube(1 TO 675)
SCREEN 1
' Draw a white box.
LINE (140, 25)-(140 + 100, 125), 3, B
' Draw the outline of a magenta cube inside the box.
DRAW "C2 BM140,50 M+50,-25 M+50,25 M-50,25"
DRAW "M-50,-25 M+0,50 M+50,25 M+50,-25 M+0,-50 BM190,75 M+0,50"
' Save the drawing in the array Cube.
GET (140, 25)-(240, 125), Cube
' Set segment to the array Cube's segment and store the drawing
' in the file MAGCUBE.GRH. Note: 2700 is the number of bytes
' in Cube (4 bytes per array element * 675).
DEF SEG = VARSEG(Cube(1))
BSAVE "MAGCUBE.GRH", VARPTR(Cube(1)), 2700
DEF SEG                    ' Restore default BASIC segment.
```

CALL Statement (BASIC Procedures)

Action Transfers control to a BASIC **SUB** procedure. To invoke a non-BASIC procedure, use the **CALL, CALLS** statements (non-BASIC).

Syntax 1 **CALL** *name* ⟦(*argumentlist*)⟧

Syntax 2 *name* ⟦(*argumentlist*)⟧

Remarks The **CALL** statement uses the following arguments:

Argument	Description
name	The name, limited to no more than 40 characters, of the BASIC **SUB** procedure being called.
argumentlist	The variables or constants passed to the procedure. Arguments in the list are separated by commas. Arguments are passed by reference (which means they can be changed by the procedure).

If *argumentlist* includes an array argument, the array is specified by the array name followed by empty parentheses:

```
DIM IntArray(1 TO 20)
.
.
.
CALL ShellSort(IntArray())
```

If you omit the optional **CALL** keyword, you must declare the procedure in a **DECLARE** statement and omit the parentheses around *argumentlist*. For more information, see Chapter 2, "SUB and FUNCTION Procedures" in the *Programmer's Guide*.

The **CALL** statement passes arguments only by reference, which means that the procedure is given the address of the argument. This allows procedures to change the argument values.

Although the **CALL** statement does not pass arguments by value, you can simulate that by enclosing the argument in parentheses. This makes the argument an expression. (If an argument is passed by value, the procedure is given the value of the argument.)

For example, the following statement calls a procedure and passes a single argument:

```
CALL SolvePuzzle((StartValue))
```

Because `StartValue` is in parentheses, BASIC evaluates it as an expression. The result is stored in a temporary location, and the address of the temporary location is passed to the **SUB** procedure. Any change made by the procedure `SolvePuzzle` is made to the temporary location only, and not to `StartValue` itself.

See Also **Absolute** Routine; **CALL, CALLS** (Non-BASIC); **DECLARE** (BASIC)

Example The following example uses the **CALL** statement to call a **SUB** procedure that prints a message on the 25th line of the display:

```
CLS       ' Clear screen.
message$ = "The quick brown fox jumped over the lazy yellow dog."
CALL PrintMessage(message$)
END

SUB PrintMessage (message$)
    LOCATE 24, 1' Move cursor to 25th line of display.
    PRINT message$;
END SUB
```

CALL, CALLS Statements (Non-BASIC Procedures)

Action Transfer control to a procedure written in another language. To invoke a BASIC procedure, use the **CALL** statement (BASIC).

Syntax 1 **CALL** *name* ⟦(*call-argumentlist*)⟧

Syntax 2 *name* ⟦*call-argumentlist*⟧

Syntax 3 **CALLS** *name* ⟦(*calls-argumentlist*)⟧

Remarks The **CALL** keyword is optional when you use the **CALL** statement. When you omit **CALL**, you must declare the procedure in a **DECLARE** statement. When you omit **CALL**, you omit the parentheses around the argument list. For more information about invoking procedures without the **CALL** keyword, see Chapter 2, "SUB and FUNCTION Procedures" in the *Programmer's Guide*.

CALLS is the same as using **CALL** with a **SEG** before each argument: every argument in a **CALLS** statement is passed as a segmented address.

The **CALL** and **CALLS** statements use the following arguments:

Argument	Description
name	The name of the non-BASIC procedure being called.
call-argumentlist	The variables, arrays, or expressions passed to the procedure.
calls-argumentlist	The variables or expressions **CALLS** passes to the procedure.

The *call-argumentlist* has two forms of syntax. In the first form, *argument* is a variable:

⟦⟦{**BYVAL**|**SEG**}⟧*argument*⟧ ⟦,⟦{**BYVAL**|**SEG**}⟧*argument*⟧...

If *argument* is an array, parentheses are required:

⟦*argument*⟦( )⟧⟧ ⟦,*argument* ⟦( )⟧⟧...

The following list describes the parts of *call-argumentlist*:

Part	Description
BYVAL	Indicates the argument is passed by value rather than by near reference (the default is by reference). **BYVAL** cannot be used if the argument is an array.
SEG	Passes the argument as a segmented (far) address. **SEG** cannot be used if the argument is an array.
argument	A BASIC variable, array, or expression passed to a procedure.

Use the first syntax if *argument* is not an array; use the second if *argument* is an array. The array is specified by the array name and a pair of parentheses. For example:

```
DIM IntArray(20) AS INTEGER
.
.
.
CALL ShellSort(IntArray) AS INTEGER
```

The argument *calls-argumentlist* has the following syntax, where *argument* is a BASIC variable or expression:

⟦*argument*⟧ ⟦,*argument*⟧...

These arguments are passed by reference as far addresses, using the segment and offset of the variable. Whole array arguments cannot be passed by **CALLS**.

The result of the **BYVAL** keyword differs from BASIC's pass by value:

```
CALL Difference (BYVAL A,(B))
```

For the first argument, only the *value* of `A` is passed to `Difference`. In contrast, `(B)` is evaluated, a temporary location is created for the value, and the address of the temporary location is passed to `Difference`.

You can use BASIC's pass by value for an argument, but the procedure in the other language must accept an address (because the address of the temporary location is passed).

Note If the argument *name* refers to an assembly language procedure, it must be a name declared with **PUBLIC** (symbol).

Be careful using the **SEG** keyword to pass elements of arrays because BASIC may move arrays in memory before the called routine begins execution. Anything in an argument list that causes memory movement may create problems. You can safely pass variables using **SEG** if the **CALL** statement's argument list contains only simple variables, arithmetic expressions, or arrays indexed without the use of intrinsic or defined functions.

See Also **CALL** (BASIC); **DECLARE** (BASIC); **DECLARE** (Non-BASIC)

Example See the **VARPTR** function programming example, which uses a **CALL** statement to invoke a C function.

CCUR Function

Action Converts a numeric expression to a currency value.

Syntax **CCUR**(*numeric-expression*)

Remarks The argument *numeric-expression* can be any numeric expression.

See Also **CDBL, CINT, CLNG, CSNG**

Example The following example passes a double-precision, floating-point value to a **FUNCTION** procedure that returns the sales tax, as a currency value, for the passed value. The returned value has four digits of precision, but it is displayed in conventional currency value format with two places to the right of the decimal. **CCUR** is used to convert from double-precision to currency data type.

```
DECLARE FUNCTION GetSalesTax@ (Amount#, percent!)
' $INCLUDE: 'FORMAT.BI'
CONST percent! = 8.1
CLS    ' Clear screen.
INPUT "What is the taxable currency amount"; Amount#
PRINT
USCurFmt$ = FormatC$(GetSalesTax@(Amount#, percent!), "$#,###,###,##0.00")
SetFormatCC (49) ' West Germany
WGCurFmt$ = FormatC$(GetSalesTax@(Amount#, percent!), "#.###.###.##0,00")
WGCurFmt$ = WGCurFmt$ + " DM"

PRINT "In the United States, currency is displayed with commas as"
PRINT "numerical separators and a period serves to separate whole from"
PRINT "fractional currency amounts. In some European countries, the"
PRINT "numerical separator is a period, and the comma serves to separate"
PRINT "whole from fractional currency amounts."
PRINT
PRINT "In the US, the tax on $"; Amount#; "at"; percent!; "percent is "
PRINT "displayed as "; USCurFmt$
PRINT
PRINT "In West Germany, the tax on"; Amount#; "DM at"; percent!; "percent"
PRINT "is displayed as "; WGCurFmt$
END

FUNCTION GetSalesTax@ (Amount#, percent!)
    GetSalesTax@ = CCUR(Amount# * (percent! * .01))
END FUNCTION
```

CDBL Function

Action Converts a numeric expression to a double-precision value.

Syntax **CDBL**(*numeric-expression*)

Remarks The argument *numeric-expression* can be any numeric expression. This function has the same effect as assigning the numeric expression to a double-precision variable.

The results of **CDBL** are no more accurate than the original expression. The added digits of precision are not significant unless the expression is calculated with double-precision accuracy.

See Also **CCUR, CINT, CLNG, CSNG**

Example The following example demonstrates how the precision of the argument affects the value returned by **CDBL**:

```
X = 7/9
X# = 7/9
PRINT X
' Both X# and CDBL(X) will be accurate to only 7 decimal
' places, because 7/9 is evaluated in single precision.
PRINT X#
PRINT CDBL(X)
PRINT 7#/9#  ' Accurate to 15 decimal places.
```

Output

```
.7777778
.7777777910232544
.7777777910232544
.7777777777777778
```

CHAIN Statement

Action Transfers control from the current program to another program.

Syntax **CHAIN** *filespec$*

Remarks The argument *filespec$* is a string expression that represents the program to which control is passed. It may include a path. (With DOS 2.1 or earlier versions, you cannot use **CHAIN** unless *filespec$* provides a complete path, including drive.) Programs running within the QBX environment assume a .BAS extension (if no extension is given) and cannot chain to executable files (files with a .COM or .EXE extension). Programs running outside the BASIC environment assume a .EXE extension and cannot chain to BASIC source files (files with a .BAS extension).

You can pass variables between programs using the **COMMON** statement to set up a blank common block. For more information on passing variables, see the entry for the **COMMON** statement.

When you compile a program outside the QBX environment, the BCL70EFR.LIB object-module library does not support **COMMON**. There are two ways to use **COMMON** with chained programs outside the environment:

- Use the default (BRT70EFR.EXE for DOS and OS/2 real mode; BRT70EFR.DLL for OS/2 protected mode) by compiling the programs using the option in the Make EXE dialog box called "EXE Requiring BRT Module."
- Use BRT70EFR.LIB by compiling from the command line without the /O option. For more information, see Appendix C, "Command-Line Tools Quick Reference."

Note When programs use BRT70EFR.EXE or BRT70EFR.DLL, files are left open during chaining unless they are explicitly closed with a **CLOSE** statement. The filenames BCL70EFR.LIB, BRT70EFR.EXE, and BRT70EFR.DLL assume that you selected these compiler options when you installed BASIC: emulator floating-point math, far strings, and real mode. If you used different options, see Chapter 17, "About Linking and Libraries" in the *Programmer's Guide* for information about other compiler options.

CHAIN is similar to **RUN**. The main differences are that **RUN** closes all open files and does not support data blocks declared with **COMMON**.

BASICA BASICA assumes the extension .BAS. The current version of BASIC assumes an extension of either .BAS or .EXE, depending on whether the program is run within the QBX environment or compiled and run outside the environment. If you omit the file extension, **CHAIN** works the same in BASICA and in the current version of BASIC.

The current version of BASIC does not support the all, merge, or delete option available in BASICA, nor does it allow you to specify a line number.

Without the line-number option, execution always starts at the beginning of the chained-to program. Thus, a chained-to program that chains back to a carelessly written chaining program can cause an endless loop.

See Also **CALL** (BASIC), **COMMON**, **RUN**

Example In the following example, the **COMMON** statement is used to pass variables between two chained programs. The first program reads in a series of numeric values, stores the values in an array, then uses the **CHAIN** statement to load and run the other program. The second program finds and prints the average of the values. The first two statements initialize the variables that are passed to `PROG2`.

```
DIM values(1 TO 50)
COMMON values(), NumValues%

' PROG2 creation section.
' The following few lines of code create the second program on the
' disk so that it can load and run with the CHAIN statement. It uses
' PRINT # statements to put the proper BASIC statements in a file.
OPEN "PROG2.BAS" FOR OUTPUT AS #1
PRINT #1, "' PROG2 for the CHAIN and COMMON statements example."
PRINT #1, "DIM X(1 TO 50)"
PRINT #1, "COMMON X(), N%"
PRINT #1, "PRINT"
PRINT #1, "IF N% > 0 THEN"
PRINT #1, "   Sum% = 0"
PRINT #1, "   FOR I% = 1 TO N%"
PRINT #1, "      Sum% = Sum% + X(I%)"
PRINT #1, "   NEXT I%"
PRINT #1, "   PRINT "; CHR$(34); "The average of the values is";
PRINT #1, CHR$(34); "; Sum% / N%"
PRINT #1, "END IF"
PRINT #1, "END"
CLOSE #1
' End of PROG2 creation section.
```

```
CLS
PRINT "Enter values one per line. Type 'END' to quit."
NumValues% = 0
DO
   INPUT "-> ", N$
   IF I% >= 50 OR UCASE$(N$) = "END" THEN EXIT DO
   NumValues% = NumValues% + 1
   values(NumValues%) = VAL(N$)
LOOP
PRINT "Press any key to continue... "
DO WHILE INKEY$ = ""
LOOP
CHAIN "PROG2.BAS"
END
```

CHDIR Statement

Action Changes the current default directory for the specified drive.

Syntax **CHDIR** *pathname$*

Remarks The argument *pathname$* is a string expression identifying the directory that is to become the default directory. The *pathname$* expression must have fewer than 64 characters. It has the following syntax:

⟦*drive*:⟧⟦\⟧*directory*⟦*directory*⟧...

The argument *drive* is an optional drive specification. If you omit *drive*, **CHDIR** changes the default directory on the current drive.

CHDIR cannot be shortened to CD.

The **CHDIR** statement changes the default directory but not the default drive. For example, if the default drive is C, the following statement changes the default directory on drive D, but C remains the default drive:

```
CHDIR "D:\TMP"
```

Use **CURDIR$** to return the current directory and **CHDRIVE** to change the default drive.

See Also **MKDIR, RMDIR**

Example These statements create the directory `\HOME\SALES` on the default drive and make it the current directory:

```
MKDIR "\HOME\SALES"
CHDIR "\HOME\SALES"
```

This statement changes the current directory to `USERS` on drive B; it does not, however, change the default drive to B:

```
CHDIR "B:USERS"
```

CHDRIVE Statement

Action Changes the current drive.

Syntax **CHDRIVE** *drive$*

Remarks The argument *drive$* is a string expression that specifies a new default drive. It must correspond to an existing drive and must be in the range A to *lastdrive*, where *lastdrive* is the maximum drive letter you set in your CONFIG.SYS file.

If you supply a null argument (`""`) for *drive$*, the current drive does not change. If the argument *drive$* is a string, **CHDRIVE** uses only the first letter. If you omit *drive$*, BASIC generates the error message `Expected: Expression` (in QBX) or `String expression required` (outside of QBX).

CHDRIVE is not case sensitive. "C" is the same as "c."

See Also **CHDIR**, **CURDIR$**

Example The following example demonstrates use of the **CHDRIVE** statements. It calls a **SUB** procedure that evaluates a file specification and changes the currently logged drive if necessary.

```
DECLARE SUB ChangeDataDrive (filespec$)
ON ERROR RESUME NEXT
filespec$ = "A:"
ChangeDataDrive filespec$
filespec$ = "C:\DOS"
ChangeDataDrive filespec$

SUB ChangeDataDrive (filespec$)
    curdrive$ = LEFT$(CURDIR$, 2)        ' Get the current drive.
    ' Determine if there is a drive name on the filespec.
    IF INSTR(1, filespec$, ":") = 2 THEN
        newdrive$ = LEFT$(filespec$, 2)
    ELSE
        newdrive$ = curdrive$
    END IF
    ' Ensure that the new drive is different from the
    ' currently logged drive.
    IF newdrive$ <> curdrive$ THEN
        CHDRIVE newdrive$            ' It isn't, so change it.
        PRINT "Logged drive changed from "; curdrive$; " to "; newdrive$
    ELSE                             ' It is, so don't change it.
        PRINT "New drive same as logged drive. No change occurred."
    END IF
END SUB
```

CHECKPOINT Statement

Action Writes all open ISAM database buffers to disk in their current state.

Syntax **CHECKPOINT**

See Also **ROLLBACK, SAVEPOINT**

Example See the programming example for the **SEEKGT**, **SEEKGE**, **SEEKEQ** statements, which uses the **CHECKPOINT** statement.

CHR$ Function

Action Returns a one-character string whose ASCII code is the argument.

Syntax **CHR$**(*code%*)

Remarks **CHR$** is commonly used to send a special character to the screen or printer. For example, you can send a form feed (`CHR$(12)`) to clear the screen and return the cursor to the home position.

CHR$ also can be used to include a double quotation mark (") in a string. The following line adds a double-quotation-mark character to the beginning and the end of the string:

```
Msg$=CHR$(34)+"Quoted string"+CHR$(34)
```

See Also ASC; Appendix A, "Keyboard Scan Codes and ASCII Character Codes"

Example The following example uses **CHR$** to display graphics characters in the extended character set and uses these characters to draw boxes on the screen:

```
DEFINT A-Z
' Display two double-sided boxes.
CALL DBox(5,22,18,40)
CALL DBox(1,4,4,50)
END

' SUB procedure to display boxes.
' Parameters:
'  Urow%, Ucol%: Row and column of upper-left corner.
'  Lrow%, Lcol%: Row and column of lower-right corner.
CONST VERT=186, HORZ=205     ' Extended ASCII graphics characters.
CONST ULEFTC=201, URIGHTC=187, LLEFTC=200, LRIGHTC=188
SUB DBox (Urow%, Ucol%, Lrow%, Lcol%) STATIC
   LOCATE Urow%, Ucol% : PRINT CHR$(ULEFTC); Draw top of box.
   LOCATE ,Ucol%+1 : PRINT STRING$(Lcol%-Ucol%,CHR$(HORZ));
   LOCATE ,Lcol% : PRINT CHR$(URIGHTC);
   FOR I=Urow%+1 TO Lrow%-1                 ' Draw body of box.
      LOCATE I,Ucol% : PRINT CHR$(VERT);
      LOCATE  ,Lcol% : PRINT CHR$(VERT);
   NEXT I
   LOCATE Lrow%, Ucol% : PRINT CHR$(LLEFTC); Draw bottom of box.
   LOCATE ,Ucol%+1 : PRINT STRING$(Lcol%-Ucol%,CHR$(HORZ));
   LOCATE ,Lcol% : PRINT CHR$(LRIGHTC);
END SUB
```

CINT Function

Action Converts a numeric expression to an integer by rounding the fractional part of the expression.

Syntax **CINT**(*numeric-expression*)

Remarks If *numeric-expression* is not between –32,768 and 32,767, inclusive, BASIC generates the run-time error message `Overflow`.

CINT differs from the **FIX** and **INT** functions, which truncate, rather than round, the fractional part. See the example for the **INT** function for an illustration of the differences among these functions.

See Also **CCUR, CDBL, CLNG, CSNG, FIX, INT**

Example The following example converts an angle in radians to an angle in degrees and minutes:

```
' Set up constants for converting radians to degrees.
CONST PI=3.141593, RADTODEG=180./PI
INPUT "Angle in radians = ",Angle    ' Get the angle in radians.
Angle = Angle * RADTODEG      ' Convert radian input to degrees.
Min = Angle - INT(Angle)      ' Get the fractional part.
' Convert fraction to value between 0 and 60.
Min = CINT(Min * 60)
Angle = INT(Angle)  ' Get whole number part.
IF Min = 60 THEN    ' 60 minutes = 1 degree.
   Angle = Angle + 1
   Min = 0
END IF
PRINT "Angle equals"; Angle; "degrees"; Min; "minutes"
```

Output

```
Angle in radians = 1.5708
Angle equals 90 degrees 0 minutes
```

CIRCLE Statement

Action Draws an ellipse or circle with a specified center and radius.

Syntax **CIRCLE** ⟦**STEP** ⟧ (*x!*,*y!*),*radius!* ⟦,⟦*color&* ⟧ ⟦,⟦*start!* ⟧ ⟦,⟦*end!* ⟧ ⟦,*aspect!* ⟧⟧⟧⟧

Remarks The following list describes the parts of the **CIRCLE** statement:

Part	Description
STEP	The **STEP** option specifies that *x!* and *y!* are offsets relative to the current graphics cursor position, enabling use of relative coordinates.
x!,*y!*	The screen coordinates for the center of the circle or ellipse.
radius!	The radius of the circle or ellipse in the units of the current coordinate system, which is determined by the most recently executed **SCREEN** statement, along with any **VIEW** or **WINDOW** statements.
color&	The attribute of the desired color. See the entries for the **COLOR** and **SCREEN** statements for more information. The default color is the foreground color.
start!, *end!*	The *start!* and *end!* angles, in radians, for the arc to draw. The *start!* and *end!* arguments are used to draw partial circles or ellipses. The arguments may range in value from –2π radians to 2π radians, where π ≈ 3.141593. The default value for *start!* is 0 radians. The default value for *end!* is 2π radians.

To convert values from degrees to radians, multiply the angle (in degrees) by π/180 (which equals .0174532925199433).

If *start!* is negative, **CIRCLE** draws a radius to *start!* and treats the angle as positive. If *end!* is negative, **CIRCLE** draws a radius to *end!* and treats the angle as positive. If both *start!* and *end!* are negative, **CIRCLE** draws a radius to both *start!* and *end!* and treats the angle as positive.

The *start!* angle can be less than the *end!* angle. If you specify *end!* but not *start!*, the arc is drawn from zero to *end!*; if you specify *start!* but not *end!*, the statement draws an arc from *start!* to 2π.

aspect! — The aspect ratio, or the ratio of the *y!* radius to the *x!* radius. The default value for *aspect!* is the value required to draw a round circle in the screen mode. This value is calculated as follows:

aspect! = 4 * (*ypixels*/*xpixels*)/3

In this formula, *xpixels* by *ypixels* is the screen resolution. For example, in screen mode 1, where the resolution is 320 x 200, the default for *aspect!* is 4 * (200/320)/3, or 5/6.

If the aspect ratio is less than one, *radius!* is the *x!* radius. If aspect is greater than one, *radius!* is equal to the *y!* radius.

To draw a radius to angle 0 (a horizontal line segment to the right), do not give the angle as –0. Instead, use a very small non-zero value as shown in the following code fragment, which draws a one-quarter wedge of a circle.

```
SCREEN 2
CIRCLE (200,100),60,,-.0001,-1.57
```

You can omit an argument in the middle of the statement, but you must include the argument's comma. In the following statement, *color&* has been omitted:

```
CIRCLE STEP (150,200),94,,0.0,6.28
```

If you omit a trailing argument, do not include its corresponding comma in the statement.

The **CIRCLE** statement updates the graphics cursor position to the center of the ellipse or circle after drawing is complete.

You can use coordinates that are outside the screen or viewport. You can show coordinates as absolutes, or you can use the **STEP** option to show the position of the center point in relation to the previous point of reference. For example, if the previous point of reference were (10,10), the statement below would draw a circle with radius 75 and center offset 10 from the current x coordinate and 5 from the current y coordinate. The circle's center would be (20,15).

```
CIRCLE STEP (10,5), 75
```

Example

The following example first draws a circle with the upper-left quarter missing. It then uses relative coordinates to position a second circle within the missing quarter circle. Finally, it uses a different aspect ratio to draw a small ellipse inside the small circle.

```
CONST PI=3.141593
SCREEN 2
' Draw a circle with the upper-left quarter missing.
' Use negative numbers so radii are drawn.
CIRCLE (320,100), 200,, -PI, -PI/2
' Use relative coordinates to draw a circle within the missing
' quarter.
CIRCLE STEP (-100,-42),100
' Draw a small ellipse inside the circle.
CIRCLE STEP(0,0), 100,,,, 5/25
' Display the drawing until a key is pressed.
LOCATE 25,1 : PRINT "Press any key to end.";
DO
LOOP WHILE INKEY$=""
```

CLEAR Statement

Action Reinitializes all program variables, closes any open files, and optionally sets the stack size.

Syntax **CLEAR** [[,,*stack&*]]

Remarks The **CLEAR** statement performs the following actions:

- Closes all files and releases the file buffers.
- Clears all common variables.
- Sets numeric variables and arrays to zero.
- Sets all string variables to null.
- Reinitializes the stack and, optionally, changes its size.

The *stack&* parameter sets aside stack space for your program. BASIC takes the amount of stack space it requires, adds the number of bytes specified by *stack&*, and sets the stack size to the result.

Note Two commas are used before *stack&* to keep BASIC compatible with BASICA. BASICA included an additional argument that set the size of the data segment. Because BASIC automatically manages the data segment, the first parameter is no longer required.

If your program has deeply nested subroutines or procedures, or if you use recursive procedures, you may want to use a **CLEAR** statement to increase the stack size. You also may want to increase the stack size if your procedures have a large number of arguments.

Clearing the stack destroys the return addresses placed on the stack during a **GOSUB** operation. This makes it impossible to execute a **RETURN** statement correctly, and BASIC generates a `RETURN without GOSUB` run-time error message. If you use a **CLEAR** statement in a **SUB** or **FUNCTION** declaration, BASIC generates the run-time error message `Illegal function call.`

BASICA BASICA programs using **CLEAR** may require modification. In BASICA programs, any **DEF FN** functions or data types declared with **DEF***type* statements are lost after a **CLEAR** statement. In compiled programs, this information is not lost because these declarations are fixed at compile time.

See Also **ERASE**, **FRE**, **SETMEM**, **STACK** Function

Example The following statement clears all program variables and sets aside 2,000 bytes on the stack for the program:

```
CLEAR ,,2000
```

CLNG Function

Action Converts a numeric expression to a long (4-byte) integer by rounding the fractional part of the expression.

Syntax **CLNG**(*numeric-expression*)

Remarks If *numeric-expression* is not between –2,147,483,648 and 2,147,483,647, inclusive, BASIC generates the error message `Overflow`.

See Also **CCUR, CDBL, CINT, CSNG**

Example The following example shows how **CLNG** rounds before converting the number:

```
A=32767.45
B=32767.55
PRINT CLNG(A); CLNG(B)
```

Output

```
32767  32768
```

CLOSE Statement

Action Concludes I/O to a file, device, or ISAM table.

Syntax **CLOSE** ⟦⟦#⟧*filenumber%*⟦,⟦#⟧*filenumber%*⟧⟧...

Remarks The **CLOSE** statement complements the **OPEN** statement.

The argument *filenumber%* is the number used in the **OPEN** statement to open the file, device, or ISAM table. A **CLOSE** statement with no arguments closes all open files and devices, including all open ISAM tables.

The association of a file, device, or ISAM table with *filenumber%* ends when a **CLOSE** statement is executed. You then can reopen the file, device, or table using the same or a different file number.

Any ISAM operation that closes a table causes transactions to be committed. For example, if a type mismatch occurs while you are opening an ISAM table, the table is closed and a pending transaction is committed.

You may wish to code your programs so they first open all tables, then perform all transactions, then close tables. Make sure any operation that can close a table occurs outside a transaction.

Using a **CLOSE** statement for a file or device that was opened for sequential output writes the final buffer of output to that file or device. **CLOSE** releases all buffer space associated with the closed file, device, or table.

The **CLEAR**, **END**, **RESET**, **RUN**, and **SYSTEM** statements, and the **CHAIN** statement when used with the /O command-line option, also close all files, devices, and tables.

Note **CLOSE** commits any pending ISAM transactions.

See Also **END**, **OPEN**, **RESET**, **STOP**

Example See the **FREEFILE** function programming example, which uses the **CLOSE** statement.

CLS Statement

Action Clears various parts of the display screen.

Syntax **CLS** [[{**0** | **1** | **2**}]]

Remarks There are several ways to use the **CLS** statement, as described in the following table:

Statement	*Condition*	*Operations performed*
CLS	Graphics-screen mode with no user-defined graphics viewport	Clears default graphics viewport; regenerates bottom text line; returns cursor to the upper-left corner of the screen.
	Graphics-screen mode with user-defined graphics viewport	Clears user-defined graphics viewport; does not regenerate bottom text line; does not return the cursor to the upper-left corner of the screen.
	In screen mode 0 (text mode)	Clears text viewport; regenerates bottom text line; returns the text cursor to the upper-left corner of the screen.
CLS 0	Any screen mode, any combination of user-defined viewports	Clears the screen of all text and graphics, except bottom text line; clears graphics viewport; regenerates bottom text line; returns the cursor to the upper-left corner of the screen.
CLS 1	Graphics-screen mode with no user-defined graphics viewport	Clears default graphics viewport; does not clear text viewport; regenerates bottom text line; returns the cursor to upper-left corner of the screen.
	Graphics-screen mode with user-defined graphics viewport	Clears user-defined graphics viewport; does not clear text viewport; does not regenerate bottom text line; does not return the cursor to the upper-left corner of the screen.
	In screen mode 0 (text mode)	Has no effect.

CLS 2	Any screen mode	Clears the text viewport (even if the user-defined text viewport includes the bottom line of text, it will not be cleared); does not clear the graphics viewport; does not regenerate the bottom text line; returns the cursor to upper-left corner of the screen.

The behavior of a particular **CLS** statement depends on:

- The **CLS** statement argument: 0, 1, 2, or none.
- The current screen mode.
- Whether a **VIEW** statement has been executed that has changed the graphics viewport from the default of the entire screen.
- Whether a **VIEW PRINT** statement has been executed that established a user-defined text viewport.
- Whether a **KEY ON** statement has been executed that established the bottom text line as a list of function-key assignments. The bottom line on the screen may be 25, 30, 43, or 60, depending upon the current screen mode.

The following fundamentals about the graphics and text viewports determine how the **CLS** statement works:

- Screen mode 0 has only a text viewport. All other screen modes have both a text and graphics viewport.
- The default graphics viewport is the entire screen.
- The default text viewport is the entire screen, except for the bottom line of text.
- After a user-defined graphics viewport (or "clipping region") has been established by execution of a **VIEW** statement, it is reset to the default by execution of any of these statements: **VIEW** without arguments; **CLEAR**; any **SCREEN** statement that changes the screen resolution.
- After a user-defined text viewport has been established by execution of a **VIEW PRINT** statement, it is reset to the default by execution of either a **VIEW PRINT** statement without arguments, or any text-oriented statement, such as **WIDTH**, that changes the number of text columns or text lines on the screen.
- If the **KEY ON** statement has been executed and the **CLS** statement regenerates the bottom screen line, the **CLS** statement regenerates the function-key assignment list. If the **KEY ON** statement has not been executed, the **CLS** statement generates a line of blank characters.

See Also VIEW, VIEW PRINT, WINDOW

Example The following example draws random circles in a graphics viewport and prints to a text viewport. The graphics viewport is cleared after 30 circles have been drawn. The program clears the text viewport after printing to it 45 times.

```
RANDOMIZE TIMER
SCREEN 1
' Set up a graphics viewport with a border.
VIEW (5,5)-(100,80),3,1
' Set up a text viewport.
VIEW PRINT 12 TO 24
LOCATE 25,1 : PRINT "Press any key to stop."
' Outside the text viewport.
Count=0
DO
   CIRCLE (50,40),INT((35-4) * RND + 5),(Count MOD 4)
   ' Clear graphics viewport every 30 times.
   IF (Count MOD 30)=0 THEN CLS 1
   PRINT "Hello. ";
   ' Clear text viewport every 45 times.
   IF (Count MOD 45)=0 THEN CLS 2
   Count=Count+1
LOOP UNTIL INKEY$ <> ""
```

COLOR Statement

Action Sets the foreground and background colors for the display.

Syntax 1	COLOR [[*foreground&*]] [[, [[*background&*]] [[,*border&*]]]]	Screen mode 0 (text only)
Syntax 2	COLOR [[*background&*]] [[,*palette%*]]	Screen mode 1
Syntax 3	COLOR [[*foreground&*]]	Screen mode 4
Syntax 4	COLOR [[*foreground&*]] [[,*background&*]]	Screen modes 7-10
Syntax 5	COLOR [[*foreground&*]]	Screen modes 12,13

Remarks With the **COLOR** statement, you can set the foreground or background colors for the display. In screen mode 0, a border color also can be selected. In screen mode 1, no foreground color can be selected, but one of two three-color palettes can be selected for use with graphic statements. In screen modes 4, 12, and 13, only the foreground color can be set.

The **COLOR** statement does not determine the range of available colors. The combination of adapter, display, and screen mode determines the color range. If you use **COLOR** statement arguments outside these valid ranges, BASIC generates the error message `Illegal function call`. For more information, see the entry for the **SCREEN** statement.

The **COLOR** statement uses the following arguments:

Argument	Description
foreground&	Text color attribute in screen mode 0; text or line-drawing display color in screen mode 4; text and line-drawing color attribute in screen modes 7–10, 12, and 13.
background&	Color code or color attribute for screen background.
border&	Color code for the area surrounding the screen.
palette%	In screen mode 1, one of two three-color palettes: 0 is green, red, and brown; 1 is cyan, magenta, and bright white.

The different versions of syntax and their effects in different screen modes are described in the following table:

Screen mode	Syntax and description
0	**COLOR** ⟦*foreground&*⟧ ⟦, ⟦*background&*⟧ ⟦,*border&*⟧⟧ Modifies the current default text foreground and background colors and the screen border. The *foreground&* color value must be an integer expression between 0 and 31, inclusive. It determines the foreground color in text mode—the default color of text. Sixteen colors can be selected with the integers 0–15. A blinking version of a color is specified by adding 16 to the color number. For example, a blinking color 7 is equal to 7 + 16, or 23. The *background&* color value is an integer expression between 0 and 7, inclusive, and is the color of the background for each text character. Blinking background colors are not supported. The *border&* color, which fills the screen area outside the text area, is an integer expression between 0 and 15, inclusive. Note that the *border&* argument has no effect with the following adapters: the IBM Enhanced Graphics Adapter (EGA), the IBM Video Graphics Array adapter (VGA), and the IBM Multicolor Graphics Array adapter (MCGA).
1	**COLOR** ⟦*background&*⟧ ⟦, *palette%*⟧ The *background&* color value must be an integer expression between 0 and 3, inclusive. The argument *palette%* is an integer expression with an odd or even value between 0 and 255, inclusive. An even value makes available one set of three default foreground colors, and an odd value makes available a different default set. The default colors for *palette%* are equivalent to the following **PALETTE** statements on a system equipped with an EGA:

```
' Definition of the even palette-number set.
COLOR ,0      ' In screen mode 1, an even palette
              '   number is equivalent to:
PALETTE 1,2   ' EGA Attribute 1 = color 2 (green)
PALETTE 2,4   ' EGA Attribute 2 = color 4 (red)
PALETTE 3,6   ' EGA Attribute 3 = color 6 (yellow)

' Definition of the odd palette-number set.
COLOR ,1      ' In screen mode 1, an odd palette
              '   number is equivalent to:
PALETTE 1,3   ' EGA Attribute 1 = color 3 (cyan)
PALETTE 2,5   ' EGA Attribute 2 = color 5 (magenta)
PALETTE 3,7   ' EGA Attribute 3 = color 7 (white)
```

	Note that in screen mode 1, a **COLOR** statement overrides **PALETTE** statements that were executed previously.
4	**COLOR** ⟦*foreground&*⟧
	The screen *foreground&* color value must be an integer expression between 0 and 15, inclusive. This expression sets the display color associated with color attribute 1 only. The screen background color is fixed as black.
7–10	**COLOR** ⟦*foreground&*⟧ ⟦,*background&*⟧
	The argument *foreground&*, the line-drawing color, is a color attribute. The argument *background&*, the screen color, is a display color.
12, 13	**COLOR** ⟦*foreground&*⟧
	The argument *foreground&*, the line drawing color, is a color attribute.

BASIC generates the error message `Illegal function call` if the **COLOR** statement is used in screen modes 2, 3, or 11. Use the **PALETTE** statements to set the color in screen mode 11.

The foreground can be the same color as the background, making characters and graphics invisible. The default background color is black for all display hardware configurations and all screen modes. This background is produced by the default display color of 0 associated with the background color attribute of 0.

In screen modes 12 and 13 you can set the background color by assigning a color to attribute 0 with a **PALETTE** statement. For example, to make the background color 8224 (an olive drab), you would use the following **PALETTE** statement:

```
PALETTE 0,8224
```

To make the background color bright red-violet, use the following **PALETTE** statement:

```
PALETTE 0, 4128831
```

In screen mode 11 you can set both the foreground and background color by assigning a color to attribute 0 with a **PALETTE** statement.

With an EGA, VGA, or MCGA installed, the **PALETTE** statement gives you flexibility in assigning different display colors to the color-attribute ranges for the *foreground&*, *background&*, and *border&* colors. For more information, see the **PALETTE** statement.

See Also **PAINT**, **PALETTE**, **SCREEN** Statement

Example

The following series of examples show **COLOR** statements and their effects in the various screen modes:

```
SCREEN 0              ' Text mode only.
' Foreground& = 1 (blue), background& = 2 (green), border& = 3 (cyan).
' EGA and VGA boards do not support the border& argument.
COLOR 1, 2, 3
CLS
LOCATE 12, 25: PRINT "Press any key to continue..."
DO : LOOP WHILE INKEY$ = ""
SCREEN 1              ' Set screen to 320 x 200 graphics.
' Background& = 1 (blue).
' Foreground& = even palette number (red, green, yellow).
COLOR 1, 0
LINE (20, 20)-(300, 180), 3, B     ' Draw a brown box.
LOCATE 12, 7: PRINT "Press any key to continue..."
DO : LOOP WHILE INKEY$ = ""
' Background& = 2 (green).
' Foreground& = odd palette number (white, magenta, cyan).
COLOR 2, 1
LINE (20, 20)-(300, 180), 3, B     ' Draw a bright white box.
LOCATE 12, 7: PRINT "Press any key to continue..."
DO : LOOP WHILE INKEY$ = ""
SCREEN 0              ' Set screen to text mode.
' Foreground& = 7 (white), background& = 0 (black), border& = 0
' (black).
COLOR 7, 0, 0
CLS
END
```

COM Statements

Action Enable, disable, or suspend event trapping at the COM port.

Syntax

COM(*n%***) ON**
COM(*n%***) OFF**
COM(*n%***) STOP**

Remarks The argument *n%* specifies the number of the COM (serial) port; *n%* can be either 1 or 2.

COM(*n%***) ON** enables communications-event trapping on COM port *n%*. If a character arrives at a communications port after a **COM(***n%***) ON** statement, the routine specified in the **ON COM** statement is executed.

COM(*n%***) OFF** disables communications-event trapping on COM port *n%*. No communications trapping takes place until another **COM(***n%***) ON** statement is executed. Events occurring while trapping is off are ignored.

COM(*n%***) STOP** suspends communications-event trapping on COM port *n%*. No trapping takes place until a **COM(***n%***) ON** statement is executed. Events occurring while trapping is suspended are remembered and processed when the next **COM(***n%***) ON** statement is executed. However, remembered events are lost if **COM(***n%***) OFF** is executed

When a communications-event trap occurs (that is, the **GOSUB** operation is performed), an automatic **COM(***n%***) STOP** is executed so that recursive traps cannot take place. The **RETURN** statement from the trapping routine automatically executes a **COM(***n%***) ON** statement unless an explicit **COM(***n%***) OFF** was performed inside the routine.

For more information, see Chapter 9, "Event Handling" in the *Programmer's Guide.*

See Also **ON** *event*

Example The following example opens COM1 and then monitors the COM1 port for input. It uses the **ON COM** statement to trap communications events.

```
CLS
' Set up error handling in case COM1 doesn't exist.
ON ERROR GOTO ErrHandler
OPEN "COM1:9600,N,8,1,BIN" FOR INPUT AS #1

COM(1) ON       ' Turn on COM event processing.
ON COM(1) GOSUB Com1Handler   ' Set up COM event handling.
' Wait for a COM event to occur or a key to be pressed.
DO : LOOP WHILE INKEY$ = ""
COM(1) OFF      ' Turn off COM event handling.
CLS
END

Com1Handler:
    PRINT A$; "Something was typed on the terminal attached to COM1."
    RETURN

ErrHandler:
    SELECT CASE ERR
        CASE 68: PRINT "COM1 is unavailable on this computer.": END
        CASE ELSE: END
    END SELECT
```

COMMAND$ Function

Action Returns the command line used to invoke the program.

Syntax **COMMAND$**

Remarks The **COMMAND$** function returns the complete command line entered after your BASIC program name, including optional parameters. **COMMAND$** removes all leading blanks from the command line and converts all letters to uppercase (capital letters).

The **COMMAND$** function can be used in stand-alone executable files or, if you are executing from the QBX environment, by using the /CMD command-line option, or by selecting Modify COMMAND$ on the Run menu.

Examples The following example breaks the command line into separate arguments and stores them in an array. Each argument is separated from adjoining arguments by one or more blanks or tabs on the command line.

```
' Default variable type is integer in this module.
DEFINT A-Z

' Declare the Comline subprogram, as well as the number and
' type of its parameters.
DECLARE SUB Comline(N, A$(),Max)

DIM A$(1 TO 15)
' Get what was typed on the command line.
CALL Comline(N,A$(),10)
' Print out each part of the command line.
PRINT "Number of arguments = ";N
PRINT "Arguments are: "
FOR I=1 TO N : PRINT A$(I) : NEXT I
```

```
' SUB procedure to get command line and split into arguments.
' Parameters:  NumArgs : Number of command-line args found.
'              Args$() : Array in which to return arguments.
'              MaxArgs : Maximum number of arguments array can return.
SUB Comline(NumArgs,Args$,MaxArgs) STATIC
CONST TRUE=-1, FALSE=0
   NumArgs=0 : In=FALSE
   Cl$=COMMAND$ ' Get the command line using the COMMAND$ function.
   L=LEN(Cl$)' Go through the command line a character at a time.
   FOR I=1 TO L
      C$=MID$(Cl$,I,1)
      ' Test for character being a blank or a tab.
      IF (C$<>" " AND C$<>CHR$(9)) THEN    ' Neither blank nor tab.
         ' Test to see if you're already inside an argument.
         IF NOT In THEN' You've found the start of a new argument.
            ' Test for too many arguments.
            IF NumArgs=MaxArgs THEN EXIT FOR
            NumArgs=NumArgs+1
            In=TRUE
         END IF
         ' Add the character to the current argument.
         Args$(NumArgs)=Args$(NumArgs)+C$
      ELSE      ' Found a blank or a tab.
         ' Set "Not in an argument" flag to FALSE.
         In=FALSE
      END IF
   NEXT I
END SUB
```

The following is a sample command line and output for a stand-alone executable file (assumes program name is `arg.exe`):

```
arg one  two   three    four     five      six
```

Output

```
Number of arguments =  6
Arguments are:
ONE
TWO
THREE
FOUR
FIVE
SIX
```

COMMITTRANS Statement

Action Commits all ISAM database operations since the most recent **BEGINTRANS** statement.

Syntax **COMMITTRANS**

Remarks **COMMITTRANS** commits a pending transaction (a series of ISAM database operations).

Transactions are a way to group a series of operations so that you can commit them as a whole, rescind them all, or rescind operations since a designated savepoint. Use **BEGINTRANS** to indicate the beginning of a transaction. Use **SAVEPOINT** to designate a savepoint. Use **ROLLBACK** and **ROLLBACK ALL** to rescind all or part of a transaction's operations.

If you attempt to use **COMMITTRANS** when there is no transaction pending, BASIC generates the error message `Illegal function call.`

COMMITTRANS commits only ISAM operations. It does not affect other BASIC variables or file types.

Any ISAM operation that closes a table causes transactions to be committed. For example, if a type mismatch occurs while you are opening an ISAM table, the table is closed and a pending transaction is committed.

You may wish to code your programs so they first open all tables, then perform all transactions, then close tables. Make sure any operation that can close a table occurs outside a transaction.

See Also **BEGINTRANS, CHECKPOINT, ROLLBACK, SAVEPOINT**

Example See the **BEGINTRANS** statement programming example, which uses the **COMMITTRANS** statement.

COMMON Statement

Action Defines global variables for sharing between modules or for chaining to another program.

Syntax **COMMON** ⟦**SHARED**⟧ ⟦*/blockname/*⟧ *variablelist*

Remarks The following list describes the parts of the **COMMON** statement:

Part	Description
SHARED	An optional attribute that indicates the variables are shared with all **SUB** or **FUNCTION** procedures in the module. The **SHARED** attribute can eliminate the need for a **SHARED** statement inside **SUB** or **FUNCTION** procedures.
blockname	A valid BASIC identifier (up to 40 characters) that identifies a group of variables. Use *blockname* to share only specific groups of variables. You cannot use a type-declaration character (**%**, **&**, **!**, **#**, **@**, or **$**) in the identifier. When *blockname* is used, the **COMMON** block is a named **COMMON** block. When *blockname* is omitted, the block is a blank **COMMON** block. Items in a named **COMMON** block are not preserved across a chain to a new program. See "Using Named COMMON" and "Using COMMON with CHAIN" later in this entry.
variablelist	A list of variables to be shared between modules or chained-to programs. The same variable cannot appear in more than one **COMMON** statement in a module.

The syntax of *variablelist* is:

variable⟦( )⟧ ⟦**AS** *type*⟧ ⟦, *variable*⟦( )⟧ ⟦**AS** *type*⟧⟧...

The following list describes the parts of *variablelist*:

Part	Description
variable	Any valid BASIC variable name. It is either an array name followed by ( ), or a variable name.
AS *type*	Declares the type of the variable. The type may be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING** (for variable-length strings), **STRING** * length (for fixed-length strings), **CURRENCY**, or a user-defined type.

Note Older versions of BASIC required the number of dimensions to appear after the name of a dynamic array in a **COMMON** statement. The number of dimensions is no longer required, although BASIC accepts the older syntax to maintain compatibility with earlier versions.

A **COMMON** statement establishes storage for variables in a special area that allows them to be shared between modules or with other programs invoked with a **CHAIN** statement.

Because **COMMON** statements establish global variables for an entire program, they must appear before any executable statements. All statements are executable, except the following:

COMMON	**OPTION BASE**
CONST	**REM**
DATA	**SHARED**
DECLARE	**STATIC**
DEF*type*	**TYPE...END TYPE**
DIM (for static arrays)	All metacommands

Variables in **COMMON** blocks are matched by position and type, not by name. Therefore, variable order is significant in **COMMON** statements. In the following fragment, it is the order of the variables in the **COMMON** statements that links the variables, not the names:

```
' Main program.
COMMON A, D, E
A = 5 : D = 8 : E = 10
.
.
.
' Common statement in another module.
COMMON A, E, D       ' A = 5, E = 8, D = 10
.
.
.
```

Both static and dynamic arrays are placed in **COMMON** by using the array name followed by parentheses. The dimensions for a static array must be set with integer-constant subscripts in a **DIM** statement preceding the **COMMON** statement. The dimensions for a dynamic array must be set in a later **DIM** or **REDIM** statement. The elements of a dynamic array are not allocated in the **COMMON** block. Only an array descriptor is placed in common. For more information about static and dynamic arrays, see Appendix B, "Data Types, Constants, Variables, and Arrays" in the *Programmer's Guide.*

The size of a common area can be different from that in another module or chained program if a blank **COMMON** block has been used. When a BASIC program shares **COMMON** blocks with a routine in the user library, the calling program may not redefine the **COMMON** block to a larger size.

Errors caused by mismatched **COMMON** statements are subtle and difficult to find. An easy way to avoid mismatched **COMMON** statements is to place **COMMON** declarations in a single include file and use the **$INCLUDE** metacommand in each module.

The following program fragments show how to use the **$INCLUDE** metacommand to share a file containing **COMMON** statements among programs:

```
' This file is menu.bas.
' $INCLUDE: 'COMDEF.BI'
.
.
.
CHAIN "PROG1"
END

' This file is prog1.bas.
' $INCLUDE: 'COMDEF.BI'
.
.
.
END

' This file is COMDEF.BI.
DIM A(100),B$(200)
COMMON I,J,K,A()
COMMON A$,B$(),X,Y,Z
```

The next sections discuss using named **COMMON** blocks and using **COMMON** when chaining programs.

Using Named Common

A named **COMMON** block provides a convenient way to group variables so that different modules have access only to the common variables they need.

The following program fragment, which calculates the volume and density of a rectangular prism, uses named **COMMON** blocks to share different sets of data with two **SUB** procedures. The **SUB** procedure `Volume` needs to share only the variables representing the lengths of the sides (in **COMMON** block `Sides`). The **SUB** procedure `Density` also needs variables representing the weight (in **COMMON** block `Weight`).

```
' Main program.
DIM S(3)
COMMON /Sides/ S()
COMMON /Weight/ C
C=52
S(1)=3:S(2)=3:S(3)=6
CALL Volume
CALL Density
END

' SUB procedure Volume in a separate module.
DIM S(3)
COMMON SHARED /Sides/ S()

SUB Volume STATIC
   Vol=S(1)*S(2)*S(3)
   .
   .
   .
END SUB

' SUB procedure Density in a separate module.
DIM S(3)
COMMON SHARED /Sides/ S()
COMMON SHARED /Weight/ W
SUB Density STATIC
   Vol=S(1)*S(2)*S(3)
   Dens=W/Vol
   .
   .
   .
END SUB
```

Note

Named **COMMON** blocks are not preserved across chained programs. Use blank **COMMON** blocks to pass variables to a chained program.

Using Common with CHAIN

The **COMMON** statement provides the only way to pass the values of variables directly to a chained program. To pass variables, both programs must contain **COMMON** statements. Remember that variable order and type are significant, not variable names. The order and type of variables must be the same for all **COMMON** statements communicating between chaining programs.

Although the order and type of variables is critical for making sure the right values are passed, the **COMMON** blocks do not have to be the same size. If the **COMMON** block in the chained-to program is smaller than the **COMMON** block in the chaining program, the extra **COMMON** variables in the chaining program are discarded. If the size of the **COMMON** block in the chained-to program is larger, then additional **COMMON** numeric variables are initialized to zero. Additional string variables are initialized to null strings.

Static arrays passed in a **COMMON** block by the chaining program must be declared as static in the chained-to program. Similarly, dynamic arrays placed in common by the chaining program must be dynamic in the chained-to program.

Note When you compile a program with the /O option, common variables are not preserved across the chain. To preserve values across a chain, you must compile without the /O option (using BC) or by selecting the EXE Requiring BRT Module option in the Make EXE File dialog box (using QBX).

See Also **CALL** (BASIC), **CHAIN**, **FUNCTION**, **SUB**

Example See the **CHAIN** statement programming example, which uses the **COMMON** statement.

CONST Statement

Action Declares symbolic constants for use in place of values.

Syntax **CONST** *constantname* = *expression* ⟦,*constantname* = *expression*⟧...

Remarks The **CONST** statement uses the following arguments:

Argument	Description
constantname	A name that follows the same rules as a BASIC variable. You can add a type-declaration character (%, &, !, #, @, or $) to indicate its type, but this character is not part of the name.
expression	An expression that consists of literals (such as 1.0), other constants, or any of the arithmetic and logical operators except exponentiation (^). You also can use a single literal string such as `"Error on input"`. You cannot use string concatenation, variables, user-defined functions, or intrinsic functions—such as **SIN** or **CHR$**—in expressions assigned to constants.

If you use a type-declaration character in the name, you can omit the character when the name is used, as shown in the following example:

```
CONST MAXDIM% = 250
.
.
.
DIM AccountNames$(MAXDIM)
```

If you omit the type-declaration character, the constant is given a type based on the expression in the **CONST** statement. Strings always yield a string constant. With numeric expressions, the expression is evaluated and the constant is given the simplest type that can represent the constant. For example, if the expression gives a result that can be represented as an integer, the constant is given an integer type.

Note Names of constants are not affected by **DEF***type* statements such as **DEFINT**. A constant's type is determined either by an explicit type-declaration character or by the type of the expression.

Constants must be defined before they are referred to. The following example produces an error because the constant `ONE` is not defined before it is used to define `TWO` (constants are defined from left to right):

```
CONST TWO = ONE + ONE, ONE = 1
```

Constants declared in a **SUB** or **FUNCTION** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the module. You can use constants anywhere that you would use an expression.

A common programming practice is to use a statement such as the following (any non-zero value represents "true"):

```
TRUE=-1
```

Using constants offers several advantages over using variables for constant values:

- You only need define constants once for an entire module.
- Constants cannot be inadvertently changed.
- In stand-alone programs, using constants produces faster and often smaller code than using variables.
- Constants make programs easier to modify.

Example

The following example uses the **CONST** statement to declare symbolic constants for the ASCII values of nonprinting characters such as tab and line feed. Constants also make programs easier to modify. The program counts words, lines, and characters in a text file. A word is any sequence of nonblank characters.

```
DEFINT a-z
CONST BLANK = 32, ENDFILE = 26, CR = 13, LF = 10
CONST TABC = 9, YES = -1, NO = 0
CLS     ' Clear screen.
' Get the filename from the command line.
FileName$=COMMAND$
IF FileName$="" THEN
   INPUT "Enter input filename: ",FileName$
   IF FileName$="" THEN END
END IF
```

```
OPEN FileName$ FOR INPUT AS #1
Words=0
Lines=0
Characters=0
' Set a flag to indicate you're not looking at a
' word, then get the first character from the file.
InaWord=NO
DO UNTIL EOF(1)
   C=ASC(INPUT$(1,#1))
   Characters=Characters+1
   IF C=BLANK or C=CR or C=LF or C=TABC THEN
      ' If the character is a blank, carriage return,
      ' line feed, or tab, you're not in a word.
      IF C=CR THEN Lines=Lines+1
      InaWord=NO
   ELSEIF InaWord=NO THEN
      ' The character is a printing character,
      ' so this is the start of a word.
      InaWord=YES Count the word and set the flag.
      Words=Words+1
   END IF
LOOP
PRINT Characters, Words, Lines
END
```

COS Function

Action Returns the cosine of an angle given in radians.

Syntax **COS**(*x*)

Remarks The argument *x* can be of any numeric type.

The cosine of an angle in a right triangle is the ratio between the length of the side adjacent to the angle and the length of the hypotenuse.

COS is calculated in single precision if *x* is an integer or single-precision value. If you use any other numeric data type, **COS** is calculated in double precision.

To convert values from degrees to radians, multiply the angle (in degrees) times $\pi/180$ (or .0174532925199433). To convert a radian value to degrees, multiply it by $180/\pi$ (or 57.2957795130824). In both cases, $\pi \approx 3.141593$.

See Also **ATN, SIN, TAN**

Example The following example plots the graph of the polar equation $r = n\theta$ for values of n from 0.1-1.1. This program uses the conversion factors $x = \cos(\theta)$ and $y = \sin(\theta)$ to change polar coordinates to Cartesian coordinates. The figure plotted is sometimes known as the Spiral of Archimedes.

```
CONST PI = 3.141593
SCREEN 1 : COLOR 7      ' Gray background.
WINDOW (-4,-6)-(8,2)    ' Define window large enough for biggest spiral.
LINE (0,0)-(2.2 * PI,0),1     ' Draw line from origin to the right.

' Draw 10 spirals.
FOR N = 1.1 TO .1 STEP -.1
   PSET (0,0)       ' Plot starting point.
   FOR Angle = 0 TO 2 * PI STEP .04
      R = N * Angle     ' Polar equation for spiral.
      ' Convert polar coordinates to Cartesian coordinates.
      X = R * COS(Angle)
      Y = R * SIN(Angle)
      LINE -(X,Y),1     ' Draw line from previous point to new point.
   NEXT Angle
NEXT N
```

CREATEINDEX Statement

Action Creates an index consisting of one or more columns of an ISAM table.

Syntax **CREATEINDEX** ⟦#⟧*filenumber%*, *indexname$*, *unique%*, *columnname$*⟦, *columnname$*⟧...

Remarks The **CREATEINDEX** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the table.
indexname$	The *indexname$* is the name of the index until the index is explicitly deleted. It follows the ISAM naming conventions (details are provided later in this entry).
unique%	A non-zero value for *unique%* indicates the index is unique—no two indexed values can be the same. A value of zero for *unique%* means the indexed values need not be unique.
columnname$	Names the column or columns to be indexed. The name follows the ISAM naming conventions. If more than one column name is given, **CREATEINDEX** defines an index based on the combination of their values. The argument *columnname$* must appear in the **TYPE** statement used when the table was created.

When you initially open a table, the current index is the NULL index. The NULL index represents the order in which records were added to the file. Once an index has been created, it can be used any number of times until it is deleted from the database.

Use **SETINDEX** to make an index the current index and impose its order on the presentation of records in a table.

Note Columns that are arrays, user-defined types, or strings longer than 255 characters cannot be indexed.

ISAM names use the following conventions:

- They have no more than 30 characters.
- They use only alphanumeric characters (A-Z, a-z, 0-9).
- They begin with an alphabetic character.
- They include no BASIC special characters.

See Also **DELETEINDEX**, **GETINDEX$**, **SETINDEX**

Example This example uses the **CREATEINDEX**, **SETINDEX** and **DELETEINDEX** statements to index an ISAM file in several ways. It displays records from the file using the **GETINDEX$** and **BOF** functions and the **MOVEFIRST**, **MOVELAST**, **MOVENEXT**, and **MOVEPREVIOUS** statements.

The program uses a file called BOOKS.MDB, the sample ISAM file that SETUP copies to your disk.

```
DEFINT A-Z
TYPE BookRec
    IDNum AS DOUBLE                 ' Unique ID number for each book.
    Title AS STRING * 50            ' Book's title.
    Publisher AS STRING * 50        ' Book's publisher.
    Author AS STRING * 36           ' Book's author.
    Price AS CURRENCY               ' Book's price.
END TYPE

DIM Library AS BookRec              ' Record structure variable.
DIM msgtxt AS STRING

CONST Database = "BOOKS.MDB"        ' Name of the disk file.
CONST TableName = "BooksStock"      ' Name of the table.
tablenum = FREEFILE                 ' File number.

OPEN Database FOR ISAM BookRec TableName AS tablenum
CREATEINDEX tablenum, "A", 0, "Author"
CREATEINDEX tablenum, "I", 1, "IDnum"
CREATEINDEX tablenum, "T", 0, "Title"
CREATEINDEX tablenum, "C", 0, "Price"

' Display static instructions.
CLS : LOCATE 13, 30
PRINT "Choose a key:"
PRINT SPC(9); "Move to:"; TAB(49); "Order by: X"
PRINT : PRINT SPC(9); "F - First record"; TAB(49); "A - Author"
PRINT : PRINT SPC(9); "L - Last record"; TAB(49); "I - ID Number"
PRINT : PRINT SPC(9); "N - Next record"; TAB(49); "T - Title"
PRINT : PRINT SPC(9); "P - Previous record"; TAB(49); "C - Cost"
PRINT : PRINT SPC(9); "Q - Quit"; TAB(49); "X - No order"
LOCATE 3, 1: PRINT TAB(37); "Books"
PRINT STRING$(80, "-");
VIEW PRINT 5 TO 10              ' Set viewport for displaying records.age.
```

```
MOVEFIRST tablenum
DO
    ' Display current record.
    CLS
    RETRIEVE tablenum, Library
    PRINT "Author:     "; Library.Author;
    PRINT TAB(49); "ID #"; Library.IDNum
    PRINT "Title:      "; Library.Title
    PRINT "Publisher: "; Library.Publisher
    PRINT "Cost:      "; Library.Price
    PRINT SPC(30); msgtxt
    PRINT STRING$(64, "-");
    IF GETINDEX$(tablenum) = "" THEN
        PRINT STRING$(15, "-");
    ELSE
        PRINT "Index in use: "; GETINDEX$(tablenum);
    END IF

    ' Get keystroke from user.
    validkeys$ = "FLNPQATICX"
    DO
        Keychoice$ = UCASE$(INKEY$)
    LOOP WHILE INSTR(validkeys$, Keychoice$) = 0 OR Keychoice$ = ""
    msgtxt = ""

    ' Move to appropriate record, or change indexes.
    SELECT CASE Keychoice$
    CASE "F"
        MOVEFIRST tablenum
    CASE "L"
        MOVELAST tablenum
    CASE "N"
        MOVENEXT tablenum
        IF EOF(tablenum) THEN
            MOVELAST tablenum
            BEEP: msgtxt = "** At last record **"
        END IF
    CASE "P"
        MOVEPREVIOUS tablenum
        IF BOF(tablenum) THEN
            MOVEFIRST tablenum
            BEEP: msgtxt = "** At first record **"
        END IF
    CASE "Q"
        EXIT DO
```

```
    CASE ELSE            ' User chose an index.
        VIEW PRINT
        LOCATE 13, 59: PRINT Keychoice$;
        VIEW PRINT 5 TO 10
        IF Keychoice$ = "X" THEN Keychoice$ = ""
        SETINDEX tablenum, Keychoice$
        MOVEFIRST tablenum
    END SELECT
LOOP

' User wants to quit, so reset viewport, delete indexes and close files.
VIEW PRINT
DELETEINDEX tablenum, "A"
DELETEINDEX tablenum, "I"
DELETEINDEX tablenum, "T"
DELETEINDEX tablenum, "C"
CLOSE
END
```

CSNG Function

Action Converts a numeric expression to a single-precision value.

Syntax CSNG(*numeric-expression*)

Remarks The **CSNG** function has the same effect as assigning *numeric-expression* to a single-precision variable.

CSNG rounds the value, if necessary, before converting it.

See Also **CCUR, CDBL, CINT, CLNG**

Example The following example shows how **CSNG** rounds before converting the value:

```
A#=975.3421115#
B#=975.3421555#
PRINT A#; CSNG(A#); B#; CSNG(B#)
```

Output

```
975.3421115  975.3421  975.3421555  975.3422
```

CSRLIN Function

Action Returns the current line (row) position of the cursor.

Syntax **CSRLIN**

Remarks To return the current column position, use the **POS** function.

See Also **LOCATE, POS**

Example The following example uses a **SUB** procedure that prints a message on the screen without disturbing the current cursor position:

```
' Move cursor to center of screen, then print message.
' Cursor returned to center of screen.
LOCATE 12,40
CALL MsgNoMove("A message not to be missed.",9,35)
INPUT "Enter anything to end: ",A$

' Print a message without disturbing current cursor position.
SUB MsgNoMove (Msg$,Row%,Col%) STATIC
   CurRow%=CSRLIN   ' Save the current line.
   CurCol%=POS(0)   ' Save the current column.
   ' Print the message at the given position.
   LOCATE Row%,Col% : PRINT Msg$;
   ' Move the cursor back to original position.
   LOCATE CurRow%, CurCol%
END SUB
```

CURDIR$ Function

Action Returns the path currently in use for the specified drive.

Syntax **CURDIR$**[[(*drive$*)]]

Remarks The argument *drive$* is a string expression that specifies a drive.

The argument *drive$* must be in the range A to *lastdrive*, where *lastdrive* is the maximum drive letter you set in your CONFIG.SYS file.

If no drive is specified or if *drive$* is a null string, **CURDIR$** returns the path for the currently selected drive. This is similar to using the **CHDIR** command at the system prompt without specifying a path.

BASIC generates an error if the first character in *drive$* lies outside the range A to *lastdrive*, unless you are working in a networked environment. BASIC also generates an error if the first character in *drive$* corresponds to a drive that does not exist, or if the first character of *drive$* is not a letter.

CURDIR$ is not case sensitive. "C" is the same as "c."

See Also **CHDIR, CHDRIVE, DIR$**

Example See the example for the **CHDRIVE** statement, which uses the **CURDIR$** function.

CVI, CVL, CVS, CVD, and CVC Functions

Action Converts strings containing numeric values to numbers.

Syntax
CVI(*2-byte-string*)
CVL(*4-byte-string*)
CVS(*4-byte-string*)
CVD(*8-byte-string*)
CVC(*8-byte-string*)

Remarks **CVI**, **CVL**, **CVS**, **CVD**, and **CVC** are used with a **FIELD** statement to read numbers from a random-access file. These functions take strings defined in the **FIELD** statement and convert them to a value of the corresponding numeric type. The functions are the inverse of **MKI$**, **MKL$**, **MKS$**, **MKD$**, and **MKC$**.

The following table describes these string conversion functions:

Function	Description
CVI	Converts a 2-byte string created with **MKI$** back to an integer.
CVL	Converts a 4-byte string created with **MKL$** back to a long integer.
CVS	Converts a 4-byte string created with **MKS$** back to a single-precision number.
CVD	Converts an 8-byte string created with **MKD$** back to a double-precision number.
CVC	Converts an 8-byte string created with **MKC$** back to a currency number.

Note These BASIC record variables provide a more efficient and convenient way of reading and writing random-access files than do some older versions of BASIC.

See Also **FIELD**; **GET** (File I/O); **LSET**; **MKI$**, **MKL$**, **MKS$**, **MKD$**, and **MKC$**

Example

The following example illustrates the use of **CVS** and **MKS$**. The program creates a random-access file that contains numbers saved as strings. It then allows the user to view and change records.

```
' Define a user type for the data records.
TYPE Buffer
   AccName AS STRING * 25
   Check   AS STRING * 4
END TYPE
' Define a variable of the variable type.
DIM BankBuffer AS Buffer
OPEN "ACCOUNT.INF" FOR RANDOM AS #1 LEN = 29
CLS
RESTORE
READ AccName$, Check
I% = 0
DO WHILE UCASE$(AccName$) <> "END"
   I% = I% + 1
   BankBuffer.AccName = AccName$
   BankBuffer.Check = MKS$(Check)
   PUT #1, I%, BankBuffer
   READ AccName$, Check$
   IF AccName$ = "END" THEN EXIT DO
LOOP
CLOSE #1
DATA "Bob Hartzell", 300
DATA "Alice Provan", 150
DATA "Alex Landow", 75
DATA "Walt Riley", 50
DATA "Georgette Gump", 25
DATA "END", 0
```

```
OPEN "ACCOUNT.INF" FOR RANDOM AS #2 LEN = 29
FIELD #2, 25 AS AccName$, 4 AS Check$
DollarFormat$ = "$$#####.##"
DO
   PRINT
   DO
      CLS
      INPUT "Enter account # to update: ", Rec%
      GET #2, Rec%                     'Get the record
      PRINT : PRINT "This is the account for "; AccName$
      PRINT : INPUT "Is this the account you wanted"; R$
   LOOP WHILE UCASE$(MID$(R$, 1, 1)) <> "Y"
   ' Convert string to single-precision number.
   Checkamt! = CVS(Check$)
   PRINT
   PRINT "The opening balance for this account is";
   PRINT USING DollarFormat$; Checkamt!
   PRINT : PRINT "Enter the checks and cash withdrawals for this"
   PRINT "account below. Enter 0 when finished."
   PRINT
   DO
      INPUT "Enter amount -> ", Checkout!
      Checkamt! = Checkamt! - Checkout!
   LOOP UNTIL Checkout! = 0
   PRINT
   PRINT "Enter the deposits for this account below."
   PRINT "Enter 0 when finished."
   PRINT
   DO
      INPUT "Enter amount ->", Checkin!
      Checkamt! = Checkamt! + Checkin!
   LOOP UNTIL Checkin! = 0
   PRINT
   PRINT "The closing balance for this account is";
   PRINT USING DollarFormat$; Checkamt!
   ' Convert single-precision number to string.
   LSET Check$ = MKS$(Checkamt!)
   PUT #2, Rec%                            'Store the record.
   PRINT : INPUT "Update another"; R$
LOOP UNTIL UCASE$(MID$(R$, 1, 1)) <> "Y"
CLOSE #2
KILL "ACCOUNT.INF"     ' Erase file.
END
```

CVSMBF, CVDMBF Functions

Action Converts strings containing Microsoft-Binary-format numbers to IEEE-format numbers.

Syntax **CVSMBF** (*4-byte-string*)
CVDMBF (*8-byte-string*)

Remarks The **CVSMBF** and **CVDMBF** functions help you read old random-access files containing real numbers stored as strings in Microsoft Binary format. These functions convert the string read from the old file to an IEEE-format number.

Function	Description
CVSMBF	Converts *4-byte-string* containing a Microsoft-Binary-format number to a single-precision IEEE-format number.
CVDMBF	Converts *8-byte-string* containing a Microsoft-Binary-format number to a double-precision IEEE-format number.

The example below shows you how to read data from an old file by using **CVSMBF** and user-defined types for records.

See Also **FIELD**; **MKSMBF$**, **MKDMBF$**

Example The following example reads records from a random-access file containing Microsoft-Binary-format numbers stored as strings. Each record contains a student's name and a test score.

```
' Define a user type for the data records.
TYPE StudentRec
   NameField AS STRING * 20
   Score AS STRING * 4
END TYPE
' Define a variable of the user type.
DIM Rec AS StudentRec
```

```
' This part of the program creates a random-access file to be used
' by the second part of the program, which demonstrates the CVSMBF
' function.
OPEN "TESTDAT1.DAT" FOR RANDOM AS #1 LEN = LEN(Rec)
CLS
RESTORE
READ NameField$, S
I = 0
DO WHILE UCASE$(NameField$) <> "END"
   I = I + 1
   Rec.NameField = NameField$
   LSET Rec.Score = MKSMBF$(S)
   PUT #1, I, Rec
   READ NameField$, S
   IF NameField$ = "END" THEN EXIT DO
LOOP
CLOSE #1
DATA "John Simmons",100
DATA "Allie Simpson",95
DATA "Tom Tucker",72
DATA "Walt Wagner",90
DATA "Mel Zucker",92
DATA "END",0

' Open the file for input.
OPEN "TESTDAT1.DAT" FOR RANDOM AS #1 LEN = LEN(Rec)
Max = LOF(1) / LEN(Rec)
' Read and print all of the records.
FOR I = 1 TO Max
   ' Read a record into the user-type variable Rec.
   GET #1, I, Rec
   ' Convert the score from a string containing a Microsoft-
   ' Binary-format number to an IEEE-format number.
   ScoreOut = CVSMBF(Rec.Score)
   ' Display the name and score.
   PRINT Rec.NameField, ScoreOut
NEXT I
CLOSE #1
KILL "TESTDAT1.DAT"
END
```

DATA Statement

Action Stores the numeric and string constants used by a program's **READ** statements.

Syntax **DATA** *constant*⟦,*constant*⟧...

Remarks The *constant* arguments can be any valid numeric or string constant.

Names of symbolic constants (defined in a **CONST** statement) that appear in **DATA** statements are interpreted as strings, rather than as names of constants. For example, in the following program fragment, the second data item is a string, `PI`, and not the value 3.141593:

```
CONST PI=3.141593
.
.
.
DATA 2.20, PI,45,7
.
.
.
```

A **DATA** statement can contain as many constants as will fit on a line. The constants are separated by commas. You can use any number of **DATA** statements.

Note If a string constant contains commas, colons, or leading or trailing spaces, enclose the string in double quotation marks.

Null data items (indicated by a missing value) can appear in a data list as shown below:

```
DATA 1,2,,4,5
```

When a null item is read into a numeric variable, the variable has the value 0. When a null item is read into a string variable, the variable has the null string value ("").

When working in the QBX environment, **DATA** statements can be entered only in the module-level code. BASIC moves all **DATA** statements not in the module-level code to the module-level code when it reads a source file. **READ** statements can appear anywhere in the program.

DATA statements are used in the order in which they appear in the source file. You may think of the items in several **DATA** statements as one continuous list of items, regardless of how many items are in a statement or where the statement appears in the program.

You can reread **DATA** statements by using the **RESTORE** statement.

REM statements that appear at the end of **DATA** statements must be preceded by a colon. Without a colon, BASIC interprets trailing **REM** statements as string data.

See Also READ, RESTORE

Examples The first example displays the current date by converting the date returned by the **DATE$** function:

```
' Use VAL to find the month from the string returned by DATE$.
C$ = DATE$
FOR I% = 1 TO VAL(C$)
    READ Month$
NEXT
DATA January, February, March, April, May, June, July
DATA August, September, October, November, December

' Get the day.
Day$ = MID$(C$, 4, 2)
IF LEFT$(Day$, 1) = "0" THEN Day$ = RIGHT$(Day$, 1)
' Get the year.
Year$ = RIGHT$(C$, 4)

PRINT "Today is "; Month$; " "; Day$; ", "; Year$
```

Output

```
Today is September 21, 1989
```

The following example shows how null data items are handled:

```
DATA abc,,def
DATA 1,,2
READ A$, B$, C$          ' B$ = ""
PRINT A$, B$, C$
PRINT
READ A, B, C             ' B = 0
PRINT A, B, C
```

Output

```
abc                      def
 1            0           2
```

DATE$ Function

Action Returns a string containing the current date.

Syntax **DATE$**

Remarks The **DATE$** function returns a 10-character string in the form *mm-dd-yyyy*, where *mm* is the month (01–12), *dd* is the day (01–31), and *yyyy* is the year (1980–2099).

See Also **DATE$** Statement

Example Note that the **DATE$** function in the following example prints a zero in front of the month:

```
PRINT DATE$
```

Output

```
04-21-1989
```

DATE$ Statement

Action Sets the current date in your computer's system.

Syntax **DATE$** = *stringexpression$*

Remarks The **DATE$** statement is the complement of the **DATE$** function.

If you use the **DATE$** statement to set the date, the change remains in effect until you change it again or reboot your computer.

The *stringexpression$* must have one of the following forms, where *mm* (01–12) and *dd* (01–31) are the month and day, and *yy* or *yyyy* (1980–2099) is the year:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

See Also **DATE$** Function, **TIME$** Statement

Example The following example prompts you to supply the month, date, and year, then resets the system date on your computer. Note that if you change the system date, any files you create or revise will be stamped with the new date.

```
PRINT "Enter the date below (default year is 1989)."
INPUT "    Month:    ",Month$
INPUT "    Date:     ",Day$
INPUT "    Year:     ",Year$
IF Year$ = "" THEN Year$ = "89"
DATE$ = Month$ + "/" + Day$ + "/" + Year$
```

DECLARE Statement (BASIC Procedures)

Action Declares references to BASIC procedures and invokes argument type checking.

Syntax **DECLARE** {**FUNCTION** | **SUB**} *name* ⟦(⟦*parameterlist*⟧)⟧

Remarks The **DECLARE** statement uses the following arguments:

Argument	Description
name	The name that will be used to invoke the procedure. The argument *name* is limited to 40 characters. **FUNCTION** procedure names can end in one of the type-declaration characters (**%**, **&**, **!**, **#**, **@**, or **$**) to indicate the type of value returned.
parameterlist	A list of parameters that indicates the number and type of arguments to be used to call the procedure. Syntax is shown below. Only the number and type of the arguments are significant.

For calls within BASIC, the **DECLARE** statement is required only if you call **SUB** procedures without the **CALL** keyword, or if you invoke a **FUNCTION** procedure defined in another module. For more information about invoking procedures without the **CALL** keyword, see Chapter 2, "SUB and FUNCTION Procedures" in the *Programmer's Guide*.

A **DECLARE** statement also causes the compiler to check the number and type of arguments used to invoke the procedure. QBX automatically generates **DECLARE** statements when you save your program while working in the environment. The **DECLARE** statement can appear only in module-level code (not in a **SUB** or **FUNCTION** procedure) and affects the entire module.

The *parameterlist* serves as a prototype for checking the number and type of the arguments in **SUB** and **FUNCTION** procedure calls. It has the following syntax:

variable⟦**AS** *type*⟧ ⟦, *variable*⟦**AS** *type*⟧⟧...

The argument *variable* is any valid BASIC variable name. If the variable is an array, it can be followed by the number of dimensions in parentheses, as in this fragment:

```
DECLARE SUB DisplayText (A(2) AS STRING)
DIM Text$(100,5)
.
.
.
CALL DisplayText(Text$())
```

The number of dimensions is optional.

The *type* is either **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **CURRENCY**, **STRING**, or a user-defined type. Again, only the number and types of arguments are significant.

Note You cannot have fixed-length strings in **DECLARE** statements, because only variable-length strings can be passed to **SUB** and **FUNCTION** procedures. Fixed-length strings can appear in an argument list but are converted to variable-length strings before being passed.

A variable's type also can be indicated by including an explicit type character (%, &, !, #, @, or $) or by relying on the default type.

The form of the parameter list determines whether or not argument checking is done, as shown in the following list:

Declaration	Meaning
`DECLARE SUB First`	You can omit the parentheses only if the **SUB** or **FUNCTION** procedure is separately compiled. No argument checking is done.
`DECLARE SUB First ()`	`First` has no parameters. If you use arguments in a call to `First`, BASIC generates an error. An empty parameter list indicates that the **SUB** or **FUNCTION** procedure has no parameters and that argument checking should be done.
`DECLARE SUB First (X AS LONG)`	`First` has one long-integer parameter. The number and type of the arguments in each call or invocation are checked when the parameter list appears in the **DECLARE** statement.

DECLARE also can be used to declare references to procedures written in other programming languages, such as C. See the entry for the **DECLARE** statement (Non-BASIC).

See Also **CALL** (BASIC), **DECLARE** (Non-BASIC), **FUNCTION**, **SUB**

Example In this program, use of the **DECLARE** statement allows a **SUB** procedure to be invoked without using the **CALL** keyword:

```
' Generate 20 random numbers, store them in an array, and sort.
DECLARE SUB Sort (A() AS INTEGER, N AS INTEGER)
DIM Array1(1 TO 20) AS INTEGER
DIM I AS INTEGER
' Generate 20 random numbers.
RANDOMIZE TIMER

CLS
FOR I% = 1 TO 20
   Array1(I%) = INT(RND * 100)
NEXT I%

' Sort the array, calling Sort without the CALL keyword.
' Notice the absence of parentheses around the arguments in
' the call to Sort.
Sort Array1(), 20

' Print the sorted array.
FOR I% = 1 TO 20
   PRINT Array1(I%)
NEXT I%
END

' Sort subroutine.
SUB Sort (A%(), N%)
   FOR I% = 1 TO N% - 1
      FOR J% = I% + 1 TO N%
         IF A%(I%) > A%(J%) THEN SWAP A%(I%), A%(J%)
      NEXT J%
   NEXT I%
END SUB
```

DECLARE Statement (Non-BASIC Procedures)

Action Declares calling sequences for external procedures written in other languages.

Syntax **DECLARE FUNCTION** *name* [[**CDECL**]] [[**ALIAS** "*aliasname*"]] [[([[*parameterlist*]])]]

DECLARE SUB *name* [[**CDECL**]] [[**ALIAS** "*aliasname*"]] [[([[*parameterlist*]])]]

Remarks The following list describes the parts of the **DECLARE** statement:

Part	Description
FUNCTION	Indicates that the external procedure returns a value and can be used in an expression.
SUB	Indicates that the external procedure is invoked in the same way as a BASIC **SUB** procedure.
name	The name used in the BASIC program to invoke the procedure. Names can have up to 40 characters. **FUNCTION** procedure names can include an explicit type character (%, &, !, #, @, or $) indicating the type of value the procedure returns.
CDECL	Indicates that the procedure uses the C-language argument order and naming conventions. **CDECL** passes the arguments from right to left, rather than using the BASIC convention of left to right. **CDECL** also affects the name used in searches of object files and libraries. If there is no **ALIAS** clause in the **DECLARE** statement, the type-declaration character is removed from the name of the procedure, and an underscore is added to the beginning. This becomes the name used when searching libraries and external files. If **CDECL** is used with **ALIAS**, *aliasname* is used as is.
ALIAS	Indicates that the procedure being called has another name in the .OBJ or library file.
aliasname	The name the procedure has in the file or library.
parameterlist	The parameters to be passed to the invoked procedure.

The argument *parameterlist* has the following syntax:

⟦{**BYVAL**|**SEG**}⟧ *variable* ⟦**AS** *type*⟧ ⟦,⟦{**BYVAL**|**SEG**}⟧ *variable* ⟦**AS** *type*⟧⟧...

The following list describes the parts of *parameterlist*:

Part	Description
BYVAL	Indicates the parameter is passed by value, not reference. Reference is the default. **BYVAL** can be used only with **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, and **CURRENCY** types. When **BYVAL** appears in front of a parameter, the actual argument is converted to the type indicated in the **DECLARE** statement before being passed.
SEG	Indicates the parameter is passed as a segmented address (far address).
variable	A valid BASIC variable name. Only the variable's type is significant. If the variable is an array, it can be followed by the number of dimensions in parentheses (to maintain compatibility with older versions of BASIC). For example: `DECLARE SUB EigenValue (A(2) AS DOUBLE)`
AS type	Indicates the variable's type. The argument *type* can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, **CURRENCY**, **ANY**, or a user-defined type. You also can indicate the variable's type by including an explicit type character (**%**, **&**, **!**, **#**, **@**, or **$**) in the variable name or by relying on the default type. When declaring external procedures written in other languages, you can use the **ANY** keyword in the **AS** clause. **ANY** overrides type checking for that argument. You cannot use **ANY** with arguments passed by value.

When neither **BYVAL** nor **SEG** is used, arguments are passed as near addresses (offsets).

This form of the **DECLARE** statement lets you refer to procedures written in other languages. The **DECLARE** statement also causes the compiler to check the number and type of arguments used to invoke the procedure. A **DECLARE** statement can appear only in module-level code and affects the entire source file.

The form of the parameter list determines whether or not argument type checking is done, as described in the following declarations:

Declaration	Meaning
`DECLARE SUB First CDECL`	No argument checking is done when there is no parameter list.

`DECLARE SUB First CDECL ()`	`First` has no parameters. If you use arguments in a call to `First`, BASIC generates an error. Empty parentheses indicate that the **SUB** or **FUNCTION** procedure has no parameters and that argument checking should be done.
`DECLARE SUB First CDECL (X AS LONG)`	`First` takes one long-integer argument. When a parameter list appears, the number and type of arguments are checked each invocation.

A procedure in a **DECLARE** statement can be invoked without the **CALL** keyword.

Note

You cannot have fixed-length strings in **DECLARE** statements because only variable-length strings can be passed to **SUB** and **FUNCTION** procedures. Fixed-length strings can appear in an argument list but are converted to variable-length strings before being passed.

Be careful when using the **SEG** keyword to pass arrays, because BASIC may move variables in memory before the called routine begins execution. Anything in a **CALL** statement's argument list that causes memory movement may create problems. You can safely pass variables using **SEG** if the **CALL** statement's argument list contains only simple variables, arithmetic expressions, or arrays indexed without the use of intrinsic or user-defined functions.

See Also

CALL, CALLS Statements (Non-BASIC); **DECLARE** (BASIC)

Example

The following example shows a BASIC program that calls a short C function. The C program is compiled separately and stored in a Quick library or explicitly linked to form the .EXE file.

```
DEFINT A-Z
DECLARE FUNCTION addone CDECL (BYVAL n AS INTEGER)
INPUT x
y = addone(x)
PRINT "x and y are "; x; y
END
```

The following function uses C argument passing and takes a single integer argument passed by value.

```
/* C function addone. Returns one more than its integer argument. */
int far addone(int n)
{
   return(n+1);
}
```

DEF FN Statement

Action Defines and names a function.

Syntax 1 **DEF FN***name*⟦(*parameterlist*)⟧ = *expression*

Syntax 2 **DEF FN***name*⟦(*parameterlist*)⟧
    ⟦*statementblock*⟧
    FN*name* = *expression*
    ⟦*statementblock*⟧
    ⟦**EXIT DEF**⟧
    ⟦*statementblock*⟧
END DEF

Remarks The **DEF FN** statement uses the following arguments:

Argument	Description

name	A legal variable name, which is always prefixed with **FN** (for example: `FNShort`). The name (including the **FN** prefix) can be up to 40 characters long and can include an explicit type-declaration character to indicate the type of value returned. Names that are the same except for the type-declaration character are distinct names. For example, the following are names of three different **DEF FN** functions:
	`FNString$`
	`FNString%`
	`FNString#`
	To return a value from a function defined by **DEF FN**, assign the value to the full function name:
	`FNString$ = "No answer."`

parameterlist	A list of variable names, separated by commas. The syntax is explained below. When the function is called, BASIC assigns the value of each argument to its corresponding parameter. Function arguments are passed by value. Functions defined by **DEF FN** do not accept arrays, records, or fixed-length strings as arguments.

expression	In both versions of syntax, *expression* is evaluated and the result is the function's value. In Syntax 1, *expression* is the entire body of the function and is limited to one logical line. When no expression is assigned to the name, the default return values are 0 for a numeric **DEF FN** function, and the null string (" ") for a string **DEF FN** function.

The argument *parameterlist* has the following syntax:

variable ⟦**AS** *type*⟧ ⟦, *variable* ⟦**AS** *type*⟧⟧...

Argument	Description
variable	Any valid BASIC variable name.
AS *type*	The argument *type* is **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **CURRENCY**, or **STRING**. You also can indicate a variable's type by including a type-declaration character (**%**, **&**, **!**, **#**, **@**, or **$**) in the name.

Note The **FUNCTION** procedure offers greater flexibility and control than the **DEF FN** statement. For more information, see the **FUNCTION** entry in this book and Chapter 2, "SUB and FUNCTION Procedures" in the *Programmer's Guide*.

DEF FN must define a function before the function is used. If you call a **DEF FN** function before it is defined, BASIC generates the error message `Function not defined`. **DEF FN** function definitions cannot appear inside other **DEF FN** definitions. In addition, functions defined by **DEF FN** cannot be recursive.

The **EXIT DEF** statement causes an immediate exit from the executing **DEF FN** function. Program execution continues where the **DEF FN** function was invoked.

DEF FN functions can be used only in the module in which they are defined.

A function defined by **DEF FN** can share variables with the module-level code. Variables not in *parameterlist* are global—their values are shared with the module-level code. To keep a variable value local to a function definition, declare it in a **STATIC** statement.

DEF FN can return either numeric or string values. **DEF FN** returns a string value if *name* is a string-variable name, and a numeric value if *name* is a numeric-variable name. If you assign a numeric value to a string function name or assign a string value to a numeric function name, BASIC generates the error message `Type mismatch`.

If the function is numeric, **DEF FN***name* returns a value with the precision specified by *name*. For example, if *name* specifies a double-precision variable, then the value returned by **DEF FN***name* is double precision, regardless of the precision of *expression*.

Because BASIC may rearrange arithmetic expressions for greater efficiency, avoid using functions defined by **DEF FN** that change program variables in expressions that may be reordered. The following example may give unpredictable results:

```
DEF FNShort
   I=10
   FNShort=1
END DEF
I=1 : PRINT FNShort + I + I
```

If BASIC reorders the expression so `FNShort` is called after calculating `(I+I)`, the result is 3 rather than 21. You usually can avoid this problem by isolating the **DEF FN** function call:

```
I = 1 : X = FNShort : PRINT X + I + I
```

Embedding I/O operations in **DEF FN**-defined functions used in I/O statements, or embedding graphics operations in **DEF FN**-defined functions in graphics statements, may cause similar problems.

See Also

FUNCTION, STATIC

Example

The following program uses a function defined by **DEF FN** to calculate the factorial of a number (for example, the factorial of 3 is 3 times 2 times 1):

```
DEF FNFactorial# (X%)
   STATIC Tmp#, I%
   Tmp# = 1
   FOR I% = 2 TO X%
      Tmp# = Tmp# * I%
   NEXT I%
   FNFactorial# = Tmp#
END DEF

INPUT "Enter an integer: ", Num%
PRINT : PRINT Num%; "factorial is"; FNFactorial#(Num%)
```

DEFtype Statements

Action Set the default data type for variables, **DEF FN** functions, and **FUNCTION** procedures.

Syntax

DEFINT *letterrange* ⟦, *letterrange*⟧...
DEFLNG *letterrange* ⟦, *letterrange*⟧...
DEFSNG *letterrange* ⟦, *letterrange*⟧...
DEFDBL *letterrange* ⟦, *letterrange*⟧...
DEFCUR *letterrange* ⟦, *letterrange*⟧...
DEFSTR *letterrange* ⟦, *letterrange*⟧...

Remarks The *letterrange* argument has the form:

letter1 ⟦–*letter2*⟧

The arguments *letter1* and *letter2* are any of the uppercase or lowercase letters of the alphabet. Names beginning with the letters in *letterrange* have the type specified by the last three letters of the statement: integer (**INT**), long integer (**LNG**), single precision (**SNG**), double precision (**DBL**), currency (**CUR**), or string (**STR**). For example, in the following program fragment, `Message` is a string variable:

```
DEFSTR A-Q
.
.
.
Message="Out of stack space."
```

The case of the letters in *letterrange* is not significant. These three statements are equivalent:

```
DEFINT I-N
DEFINT i-n
DEFINT i-N
```

A type-declaration character (%, &, !, #, @, or $) always takes precedence over a **DEF***type* statement. **DEF***type* statements do not affect record elements.

Note `I%`, `I&`, `I!`, `I#`, `I@`, and `I$` all are distinct variables, and each may hold a different value.

BASICA BASICA handles default data types differently. BASICA scans each statement before executing it. If the statement contains a variable without an explicit type (indicated by %, &, !, #, @, or $), the interpreter uses the current default type.

In contrast, BASIC scans the text once only and after a variable appears in a program line, its type cannot be changed.

Example See the **ABS** function programming example, which uses the **DEFDBL** statement.

DEF SEG Statement

Action Sets the current segment address for a subsequent **PEEK** function, **BLOAD**, **BSAVE**, **Absolute** routine, or **POKE** statement.

Syntax **DEF SEG** [[=*address*]]

Remarks For **Absolute**, **BLOAD**, **BSAVE**, **PEEK**, and **POKE**, *address* is used as the segment. The argument *address* is a numeric expression with an unsigned integer value between 0 and 65,535, inclusive. If you use a value outside this range, BASIC generates the error message `Illegal function call`. The previous segment is retained if an error occurs. If *address* is omitted, DGROUP is used.

DEF SEG remains in effect until changed. To reset the current segment to the default data segment (DGROUP), use **DEF SEG** without any argument.

Be sure to separate **DEF** and **SEG** with a space. Otherwise, BASIC interprets the statement to mean "assign a value to the variable `DEFSEG`."

To set the current segment address to the address of data stored in far memory, you can use **DEF SEG** with the **SSEG** or **VARSEG** functions (**SSEG** returns the current segment address of a string; **VARSEG** returns the current segment address of numeric data). For example, this statement sets the current address for a far string named `a$` :

```
DEF SEG = SSEG(a$)
```

When using **DEF SEG** in OS/2 protected mode, any **DEF SEG** statement must refer only to a valid selector. The **DEF SEG** statement itself does not generate any memory references using the selector, nor does it attempt to validate the selector. If a misdirected **DEF SEG** statement causes your program to refer to an illegal memory address, the operating system may generate a protection exception, or BASIC may generate the error message `Permission denied`. The default **DEF SEG** segment always constitutes a valid memory reference. Use caution when altering this reference in protected mode.

DEF SEG and Expanded Memory Arrays

Do not use **DEF SEG** to set the segment of an expanded memory array. If you start QBX with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays less than 16K in size.
- Fixed-length string arrays less than 16K in size.
- User-defined-type arrays less than 16K in size.

If you want to use **DEF SEG** to set the segment of an array, first start QBX without the /Ea switch. (Without the /Ea switch, no arrays are stored in expanded memory.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started.*

BASICA In this version of BASIC, the **CALL** and **CALLS** statements do not use the segment address set by **DEF SEG**.

See Also **SADD; SSEG; VARPTR, VARSEG**

Example The following example uses **DEF SEG**, **PEEK**, and **POKE** to turn the Caps Lock key on and off.

This program contains hardware-specific instructions. It works correctly on IBM PC, XT, and AT computers.

```
DECLARE SUB CapsOn ()
DECLARE SUB CapsOff ()
DECLARE SUB PrntMsg (R%, C%, M$)
CLS
CapsOn
PrntMsg 24, 1, "<Caps Lock On>"
LOCATE 12, 10
LINE INPUT "Enter a string (all characters are caps): ", S$
CapsOff
PrntMsg 24, 1, "              "
PrntMsg 25, 1, "Press any key to continue..."
DO WHILE INKEY$ = "": LOOP
CLS
END

SUB CapsOff STATIC  ' Turn the Caps Lock key off.
   DEF SEG = 0      ' Set segment to low memory.
      POKE &H417, PEEK(&H417) AND &HBF     ' Turn off bit 6 of &H0417.
   DEF SEG          ' Restore segment.
END SUB

SUB CapsOn STATIC   ' Turn Caps Lock on.
   DEF SEG = 0      ' Set segment to low memory.
      POKE &H417, PEEK(&H417) OR &H40      ' Turn on bit 6 of &H0417.
   DEF SEG          ' Restore segment.
END SUB

SUB PrntMsg (Row%, Col%, Message$) STATIC
' Print message at Row%, Col% without changing cursor.
   CurRow% = CSRLIN: CurCol% = POS(0)      ' Save cursor position.
   LOCATE Row%, Col%: PRINT Message$;
   ' Restore cursor.
   LOCATE CurRow%, CurCol%
END SUB
```

DELETE Statement

Action Removes the current record from an ISAM table.

Syntax **DELETE** [[#]]*filenumber%*

Remarks The argument *filenumber%* is the number used in the **OPEN** statement to open the table.

If the record you delete is the last record according to the table's current index, the current position is at the end of the table and there is no current record. When you delete any other record in the table, the record following the deleted record becomes current. The record preceding the deleted record remains the previous record.

See Also **DELETEINDEX, DELETETABLE, INSERT**

Example See the **BEGINTRANS** statement programming example, which uses the **DELETE** statement.

DELETEINDEX Statement

Action Permanently removes an index from an ISAM database.

Syntax **DELETEINDEX** [[#]]*filenumber%*, *indexname$*

Remarks The **DELETEINDEX** statement uses the following arguments:

Argument	Description
filenumber%	The number of the open ISAM table that was used in the **OPEN** statement.
indexname$	The name used in the **CREATEINDEX** statement to create the index.

See Also **CREATEINDEX, GETINDEX$, SETINDEX**

Example See the **CREATEINDEX** statement programming example, which uses the **DELETEINDEX** statement.

DELETETABLE Statement

Action Removes a table and deletes any indexes based on the table from an ISAM database.

Syntax **DELETETABLE** *database$*,*tablename$*

Remarks The **DELETETABLE** statement uses the following arguments:

Argument	Description
database$	A database filename or path. The argument *database$* specifies an optional device, followed by a filename or path conforming to the DOS naming conventions. The *database$* specification must be the name of an ISAM file.
tablename$	The name used in the **OPEN** statement to open the table.

If the specified file is not an ISAM database file, BASIC generates the error message `Bad file mode.`

Warning Use **DELETETABLE** with caution. **DELETETABLE** permanently removes the specified table from the database.

See Also **CLOSE**, **DELETE**, **DELETEINDEX**, **OPEN**

Example See the programming example for the **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements, which uses the **DELETETABLE** statement.

DIM Statement

Action Declares a variable and allocates storage space.

Syntax **DIM** [[**SHARED**]] *variable*[[(*subscripts*)]] [[**AS** *type*]] [[,*variable*[[(*subscripts*)]] [[**AS** *type*]]]]...

Remarks The following list describes the parts of the **DIM** statement:

Part	Description
SHARED	The optional **SHARED** attribute allows all procedures in a module to share arrays and simple variables. This differs from the **SHARED** statement, which affects only variables within a single **SUB** or **FUNCTION** procedure.
variable	A BASIC variable name.
subscripts	The dimensions of the array. Multiple dimensions can be declared. The subscript syntax is described below.
AS *type*	Declares the type of the variable. The type may be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING** (for variable-length strings), **STRING** * *length* (for fixed-length strings), **CURRENCY**, or a user-defined type.

The argument *subscript* in the **DIM** statement has the following form:

[[*lower%* **TO**]] *upper%* [[,[[*lower%* **TO**]] *upper%*]]...

The **TO** keyword provides a way to indicate both the lower and the upper bounds of an array's subscripts. The following statements are equivalent (if there is no **OPTION BASE** statement):

```
DIM A(8,3)
DIM A(0 TO 8, 0 TO 3)
DIM A(8,0 TO 3)
```

Subscripts can be negative. **TO** can be used to specify any range of subscripts between –32,768 and 32,767, inclusive:

```
DIM A(-4 TO 10)
DIM B(-99 TO -5,-3 TO 0)
```

If you use an implicitly dimensioned array (that is, any array that has not been declared with the **DIM** statement), the maximum value of each subscript in the array is 10. If you use a subscript that is greater than the specified maximum and if run-time error checking is in effect, BASIC generates the error message `Subscript out of range.`

Note

Run-time error checking is in effect when:

- You run a program from the QBX environment.
- You have created an .EXE file in the QBX environment and you have selected the Run-Time Error Checking option in the Make EXE File dialog box.
- You have compiled your program with BC using the /D option to select run-time error checking.

The **DIM** statement initializes all elements of numeric arrays to zero and all the elements of string arrays to null strings. The fields of record variables are initialized to zero, including fixed-string fields. The maximum number of dimensions allowed in a **DIM** statement is 60.

If you try to declare a dimension for an array variable with a **DIM** statement after you have referred to the array, BASIC generates the error message `Array already dimensioned`. It is good programming practice to put the required **DIM** statements at the beginning of a program, outside of any loops.

Static and Dynamic Arrays

How you declare an array also determines whether it is static (allocated when the program is translated) or dynamic (allocated when the program is run):

How array is declared	Allocation
Declared first in a **COMMON** statement	Dynamic
Implicitly dimensioned arrays	Static
Dimensioned with numeric constants or **CONST** statement constants	Static
Dimensioned with variables as subscripts	Dynamic

The following list shows examples of different **DIM** statements and results:

Statement	Result
`DIM A(0 TO 9)`	The array `A` is allocated as a static array if **$DYNAMIC** is not in effect.
`DIM A(MAXDIM)`	If `MAXDIM` is defined in a **CONST** statement, `A` is a static array. If `MAXDIM` is a variable, the array is a dynamic array and is allocated only when the program reaches the **DIM** statement.

For more information about static and dynamic arrays, see Appendix B, "Data Types, Constants, Variables, and Arrays" in the *Programmer's Guide*.

Note If the array size exceeds 64K, if the array is not dynamic, and if the /AH option was not used, BASIC may generate the error message `Subscript out of range` or `Array too big`. Reduce the size of the array or make the array dynamic and use the /AH command-line option.

Type Declarations

In addition to declaring the dimensions of an array, the **DIM** statement also can be used to declare the type of a variable. For example, the following statement declares the variable to be an integer, even though there is no type-declaration character or **DEFINT** statement:

```
DIM NumberOfBytes AS INTEGER
```

The **DIM** statement provides a way to declare specific variables to be records. In the following example, the variable `TopCard` is declared as a record variable:

```
TYPE Card
   Suit AS STRING * 9
   Value AS INTEGER
END TYPE

DIM TopCard AS Card
```

You also can declare arrays of records:

```
TYPE Card
   Suit AS STRING * 9
   Value AS INTEGER
END TYPE

DIM Deck(1 TO 52) AS Card
```

Note BASIC now supports the **CURRENCY** data type (type suffix @). This can be used in the **AS** *type* clause of the **DIM** statement. Also, BASIC now supports static arrays in user-defined types.

BASICA BASICA executes a **DIM** statement when it encounters the statement in the program. The array is allocated only when the statement is executed, so all arrays in BASICA are dynamic.

See Also **ERASE, OPTION BASE, REDIM**

Example

The following example finds and prints the maximum and minimum of a set of values:

```
' Declare the dimensions for an array to hold the values.
CONST MAXDIM=20
DIM A(1 TO MAXDIM)
' Use DIM to set up two integer variables. Other variables are SINGLE.
DIM NumValues AS INTEGER, I AS INTEGER

' Get the values.
NumValues=0
PRINT "Enter values one per line. Type END to end."
DO
   INPUT A$
   IF UCASE$(A$)="END" OR NumValues>=MAXDIM THEN EXIT DO
   NumValues=NumValues+1
   A(NumValues)=VAL(A$)
LOOP

' Find the maximum and minimum values.
IF NumValues>0 THEN
   Max=A(1)
   Min=A(1)
   FOR I=1 TO NumValues
      IF A(I)>Max THEN Max=A(I)
      IF A(I)<Min THEN Min=A(I)
   NEXT I
   PRINT "The maximum is ";Max;" The minimum is ";Min
ELSE
   PRINT "Too few values."
END IF
```

Output

```
Enter values one per line. Type END to end.
? 23.2
? 11.3
? 1.6
? end
The maximum is  23.2  The minimum is  1.6
```

DIR$ Function

Action Returns a filename that matches the specified pattern.

Syntax **DIR$** [[(*filespec$*)]]

Remarks The argument *filespec$* is a string expression that specifies a filename or path.

The path and filename can include a drive and DOS wildcard characters.

BASIC generates the error message `Illegal function call` if you don't specify *filespec$* when you first call **DIR$**.

DIR$** returns the first filename that matches *filespec$*. To retrieve additional filenames that match the *filespec$* pattern, call **DIR$ again with no argument. When no filenames match, **DIR$** returns a null string.

You do not have to retrieve all of the filenames that match a given *filespec$* before calling **DIR$** again with a new *filespec$*.

Because filenames are retrieved in no particular order, you may want to store filenames in a dynamic array and sort the array.

DIR$ is not case sensitive. "C" is the same as "c."

See Also **CURDIR$**

Example The following example demonstrates use of the **DIR$** function:

```
DECLARE FUNCTION GetFileCount& (filespec$)
filespec$ = "*.*"
count = GetFileCount(filespec$)
PRINT count; "files match the file specification."

' Function that returns number of files that match file specification.
FUNCTION GetFileCount& (filespec$)
DIM FileCount AS LONG
    IF LEN(DIR$(filespec$)) = 0 THEN     ' Ensure filespec$ is valid.
        FileCount& = 0
    ELSE
        FileCount = 1
        DO WHILE LEN(DIR$) > 0
            FileCount& = FileCount& + 1
        LOOP
    END IF
    GetFileCount = FileCount&
END FUNCTION
```

DO...LOOP Statement

Action Repeats a block of statements while a condition is true or until a condition becomes true.

Syntax 1

DO ⟦{**WHILE** | **UNTIL**} *condition*⟧
  ⟦*statementblock*⟧
  ⟦**EXIT DO**⟧
  ⟦*statementblock*⟧
LOOP

Syntax 2

DO
  ⟦*statementblock*⟧
  ⟦**EXIT DO**⟧
  ⟦*statementblock*⟧
LOOP ⟦{**WHILE** | **UNTIL**} *condition*⟧

Remarks The argument *condition* is a numeric expression that BASIC evaluates as true (nonzero) or false (zero).

The program lines between the **DO** and **LOOP** statements will be repeated as long as *condition* is true.

You can use a **DO...LOOP** statement instead of a **WHILE...WEND** statement. The **DO...LOOP** is more versatile because it can test for a condition at the beginning or at the end of a loop.

Examples

The following examples show how placement of the condition affects the number of times the block of statements is executed:

```
' Test at the beginning of the loop. Because I is not less than 10,
' the body of the loop (the statement block) is never executed.
I = 10
PRINT "Example 1:": PRINT
PRINT "Value of I at beginning of loop is  "; I
DO WHILE I < 10
   I = I + 1
LOOP
PRINT "Value of I at end of loop is  "; I

' Test at the end of the loop, so the statement block executes
' at least once.
I = 10
PRINT : PRINT : PRINT "Example 2:": PRINT
DO
   PRINT "Value of I at beginning of loop is  "; I
   I = I + 1
LOOP WHILE I < 10
PRINT "Value of I at end of loop is  "; I
```

The following sort program tests at the end of the loop because the entire array must be examined at least once to see if it is in order. The program illustrates testing at the end of a loop:

```
CONST NOEXCH = -1   ' Set up a value to indicate no exchanges.
DIM Exes(12)
DIM I AS INTEGER
' Load the array and mix it up.
FOR I = 1 TO 12 STEP 2
   Exes(I) = 13 - I
   Exes(I + 1) = 0 + I
NEXT I
Limit = 12
PRINT
PRINT "This is the list of numbers to sort in ascending order:"
PRINT
FOR I = 1 TO 12
   PRINT USING " ### "; Exes(I);
NEXT I
PRINT
```

```
' In the following DO...LOOP, INKEY$ is tested at the bottom of
' the loop. When the user presses a key, INKEY$ is no longer a
' null string and the loop terminates, continuing program execution.
PRINT : PRINT "Press any key to continue."
DO
LOOP WHILE INKEY$ = ""

DO
   Exchange = NOEXCH
   FOR I = 1 TO Limit - 1            ' Make one pass over the array.
      IF Exes(I) > Exes(I + 1) THEN
         SWAP Exes(I), Exes(I + 1)   ' Exchange array elements.
         Exchange = I                ' Record location of most
      END IF                         ' recent exchange.
   NEXT I
   Limit = Exchange                  ' Sort on next pass only to where
                                     ' last exchange was done.
LOOP UNTIL Exchange = NOEXCH         ' Sort until no elements are
                                     ' exchanged.
PRINT : PRINT "Sorting is completed. This is the sorted list:": PRINT
FOR I = 1 TO 12
   PRINT USING " ### "; Exes(I);
NEXT I
END
```

DRAW Statement

Action Draws an object described by a string of drawing commands.

Syntax **DRAW** *stringexpression$*

Remarks The argument *stringexpression$* contains one or more drawing commands. The drawing commands combine many of the capabilities of the other graphics statements (such as **LINE** and **COLOR**) into a graphics macro language, as described below.

There are three types of drawing commands:

- Line-drawing and cursor-movement commands.
- Rotation, color, and scale-factor commands.
- The substring command.

Cursor-Movement Commands

The following prefix commands can precede any of the movement commands:

Prefix	*Description*
B	Move, but do not plot any points.
N	Plot, but return to original position when done.

The following commands specify movement in terms of units. The default unit size is one pixel. Unit size can be modified by the S command, which sets the scale factor (see "Rotation, Color, and Scale-Factor Commands" later in this entry). If no unit argument is supplied, the graphics cursor is moved one unit.

Each of the cursor-movement commands initiate movement from the current graphics position, which is usually the coordinate of the last graphics point plotted with a **DRAW** macro-language command or another graphics command (such as **LINE** or **PSET**). The current position defaults to the center of the screen when a program begins execution. The cursor-movement commands have the following effects:

Command	Effects
U [[*n*]]	Move up *n* units.
D [[*n*]]	Move down *n* units.
L [[*n*]]	Move left *n* units.
R [[*n*]]	Move right *n* units.
E [[*n*]]	Move diagonally up and right *n* units.
F [[*n*]]	Move diagonally down and right *n* units.
G [[*n*]]	Move diagonally down and left *n* units.
H [[*n*]]	Move diagonally up and left *n* units.
M [[{+|–}]]*x*,*y*	Move absolute or relative: If *x* is preceded by a plus (+) or minus (–), the movement is relative to the current point; that is, *x* and *y* are added to (or subtracted from) the coordinates of the current graphics position and movement is to that point. If no sign precedes *x*, the movement is absolute. Movement is from the current cursor position to the point with coordinates *x*,*y*.

Rotation, Color, and Scale-Factor Commands

The following commands let you change the appearance of a drawing by rotating it, changing colors, or scaling it:

Command	Description
A *n*	Set angle rotation. The value of *n* may range from 0 to 3, where 0 is 0°, 1 is 90°, 2 is 180°, and 3 is 270°. Figures rotated 90° or 270° are scaled so they appear the same size on a monitor screen with a 4:3 aspect ratio.

TA *n*	Turn an angle of *n* degrees. The value of *n* must be between –360 and 360. If *n* is positive, rotation is counterclockwise; if *n* is negative, rotation is clockwise. The following example uses TA to draw spokes:
	`SCREEN 1`
	`FOR D = 0 TO 360 STEP 10`
	`  DRAW "TA=" + VARPTR$(D) + "NU50"`
	`NEXT D`

C *n*	Set the drawing (foreground) color to *n*. See the **COLOR**, **PALETTE**, and **SCREEN** statements for discussion of valid colors, numbers, and attributes.
P *p*, *b*	The value *p* is the paint color for the figure's interior, while *b* is the paint color for the figure's border. See the **PAINT** statement for more information about painting an area with a graphic pattern.
S *n*	Set scale factor *n*, which can range from 0 to 255, inclusive. Increase or decrease length of moves. The default for *n* is 4, which causes no scaling. The scale factor multiplied by movement-command arguments (divided by 4) gives the actual distance moved.
X *stringexpression$*	Execute substring. This command allows you to execute a second substring from a **DRAW** command string. You can have one string expression execute another, which executes a third, and so on. Numeric arguments to macro commands within *stringexpression$* can be constants or variable names. BASIC requires the following syntax: "X" + **VARPTR$**(*stringexpression*)

BASICA

Some **DRAW** statements that are allowable in BASICA programs require modification when used with the BASIC compiler. Specifically, the compiler requires the **VARPTR$** form for variables. One example is this BASICA statement (in which `ANGLE` is a variable):

```
DRAW "TA = Angle"
```

For the BASIC compiler, you would change that to:

```
DRAW "TA =" + VARPTR$(Angle)
```

The compiler does not support the BASICA X *stringexpression$* command. However, you can execute a substring by appending the character form of the address to X. For example, the following two statements are equivalent. The first statement works when within the environment and when using the compiler, while the second works only within the QBX environment.

```
DRAW "X" + VARPTR$(A$)
DRAW "XA$"
```

See Also

PALETTE, **SCREEN** Statement, **VARPTR$**

Examples

The first example draws the outline of a triangle in magenta and paints the interior cyan:

```
SCREEN 1
DRAW "C2"                ' Set color to magenta.
DRAW "F60 L120 E60"      ' Draw a triangle.
DRAW "BD30"              ' Move down into the triangle.
DRAW "P1,2"              ' Paint interior.
```

The next example shows how to use the M macro command with absolute and relative movement, and with string- and numeric-variable arguments:

```
SCREEN 2
PRINT "Press any key to continue..."

' Absolute movement.
DRAW "M 50,80"
DRAW "M 80,50"
LOCATE 2, 30: PRINT "Absolute movement"
DO : LOOP WHILE INKEY$ = ""

' Relative movement.
DRAW "M+40,-20"
DRAW "M-40,-20"
DRAW "M-40,+20"
DRAW "M+40,+20"
LOCATE 3, 30: PRINT "Relative movement"
DO : LOOP WHILE INKEY$ = ""
```

```
' Using a string variable.
X$ = "400": Y$ = "190"
DRAW "M" + X$ + "," + Y$
LOCATE 4, 30: PRINT "String variable"
DO : LOOP WHILE INKEY$ = ""

' Using numeric variables (note the two "=" signs).
A = 300: B = 120
DRAW "M=" + VARPTR$(A) + ",=" + VARPTR$(B)
LOCATE 5, 30: PRINT "Numeric variables"
```

This program draws a clock on the screen using the **TIME$** function:

```
' Declare procedure.
DECLARE SUB Face (Min$)

' Select 640 x 200 pixel high-resolution graphics screen.
SCREEN 2
DO
   CLS
   Min$ = MID$(TIME$, 4, 2)   ' Get string containing minutes value.
   Face Min$                  ' Draw clock face.
   ' Wait until minute changes or a key is pressed.
   DO
      ' Print time at top of screen.
      LOCATE 2, 37
      PRINT TIME$
      Test$ = INKEY$          ' Test for a key press.
   LOOP WHILE Min$ = MID$(TIME$, 4, 2) AND Test$ = ""
LOOP WHILE Test$ = ""         ' End program when a key is pressed.
END

SUB Face (Min$) STATIC ' Draw the clock face.
   LOCATE 23, 30
   PRINT "Press any key to end"
   CIRCLE (320, 100), 175
   ' Convert strings to numbers.
   Hr = VAL(TIME$)
   Min = VAL(Min$)
   ' Convert numbers to angles.
   Little = 360 - (30 * Hr + Min / 2)
   Big = 360 - (6 * Min)
   ' Draw the hands.
   DRAW "TA=" + VARPTR$(Little) + "NU40"
   DRAW "TA=" + VARPTR$(Big) + "NU70"
END SUB
```

END Statement

Action Stops a BASIC program, procedure, or block.

Syntax 1 **END** [[{**DEF** | **FUNCTION** | **IF** | **SELECT** | **SUB** | **TYPE**}]]

Syntax 2 **END** [[*n%*]]

Remarks There are a number of ways to use the **END** statement, as described in the following list:

Statement	Description
END DEF	Ends a multiline **DEF FN** function definition. You must use **END DEF** with a multiline **DEF FN**.
END FUNCTION	Ends a **FUNCTION** procedure definition. You must use **END FUNCTION** with **FUNCTION**.
END IF	Ends a block **IF...THEN...ELSE** statement. You must use **END IF** with block **IF...THEN...ELSE**.
END SELECT	Ends a **SELECT CASE** block. You must use **END SELECT** with a **SELECT CASE** statement.
END SUB	Ends a BASIC **SUB** procedure. You must use **END SUB** with **SUB**.
END TYPE	Ends a user-defined type definition (**TYPE** statement). You must use **END TYPE** with **TYPE**.
n%	Ends a program and returns the value *n%* to the operating system (if *n%* is omitted, the value of *n%* is set to 0). **END** is not required at the end of a program. The value *n%* can be used by DOS or OS/2 batch files or by non-BASIC programs. Untrapped errors and fatal errors set the value of *n%* to –1.

By itself, the **END** statement stops program execution and closes all files. In a stand-alone program, **END** returns control to the operating system. In the QBX environment, **END** returns to that environment. You can place an **END** statement anywhere in the program to terminate execution.

See Also **DEF FN, FUNCTION, IF...THEN...ELSE, SELECT CASE, STOP, SUB, SYSTEM, TYPE**

Example See the **ON ERROR** statement programming example, which uses the **END** statement.

ENVIRON$ Function

Action Returns an environment string from the DOS or OS/2 environment-string table.

Syntax 1 **ENVIRON$** (*environmentstring$*)

Syntax 2 **ENVIRON$** (*n%*)

Remarks The **ENVIRON$** function uses the following arguments:

Argument	Description
environmentstring$	A string constant or variable that contains the name of an environment variable. The name of the environment variable must be uppercase. For example, `ENVIRON$ ("PATH")` returns the path environment variable; `ENVIRON$ ("path")` returns a null string.
n%	A numeric expression that indicates that the *n*th string from the environment string table should be returned.

If you specify an environment-string name, but it cannot be found in the environment-string table, or there is no text following it, then **ENVIRON$** returns a null string. Otherwise, **ENVIRON$** returns the text following the equal sign in the environment-string table.

If you specify a numeric argument (*n%*), the *n*th string in the environment-string table is returned. In this case, the string includes all of the text, including *environmentstring$*. If the *n*th string does not exist, **ENVIRON$** returns a null string. The argument *n%* can be any numeric expression; it is rounded to an integer.

See Also ENVIRON Statement

Example See the **ENVIRON** statement programming example, which uses the **ENVIRON$** function.

ENVIRON Statement

Action Modifies or adds a parameter in the DOS or OS/2 environment-string table.

Syntax **ENVIRON** *stringexpression$*

Remarks The argument *stringexpression$* is a string constant or variable that contains the name of the environment variable, such as PATH or PROMPT. It can have the form *parameterID=text*, or *parameterID text*. Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right, text.

The argument *parameterID* must contain all uppercase letters. For example:

This statement:	***Has this effect:***
`ENVIRON "PATH=C:\SALES"`	Changes the path.
`ENVIRON "path=C:\SALES"`	Does not change the path. (It creates a new environment parameter not usable by the operating system.)

If *parameterID* did not previously exist in the environment-string table, it is appended to the end of the table. If *parameterID* exists in the table when the **ENVIRON** statement is executed, it is deleted and the new *parameterID* value is appended to the end of the table.

The text string is the new parameter text. If the text is a null string (`""`) or a semicolon (`";"`), the existing parameter is removed from the environment-string table and the remaining body of the table is compressed.

DOS or OS/2 discards the environment-string table modified by this function when your program ends. The environment-string table is then the same as it was before your program ran.

Note You cannot increase the size of the environment-string table when using the **ENVIRON** statement. This means that before you can add a new environment variable or increase the size of an existing environment variable you must first delete or decrease the size of existing environment variable(s).

You can use this statement to change the PATH parameter for a "child" process (a program or command started by a **SHELL** statement) or to pass parameters to a child process by creating a new environment variable.

BASIC generates the error message `Out of memory` when no more space can be allocated to the environment-string table. The amount of free space in the table usually is quite small.

See Also **ENVIRON$**, **SHELL** Function

Example The following example uses the **ENVIRON$** function to get a copy of the current DOS PATH variable. The PATH variable is then changed using the **ENVIRON** statement. The contents of the environment-string table are then displayed using the **ENVIRON$** function.

```
DEFINT A-Z
' Initialize variables.
Path$ = "PATH="
I% = 1
' Store the old PATH.
OldPath$ = ENVIRON$("PATH")
ENVIRON "PATH=C:\BIN;C:\DOS;C:\BUSINESS\ACCOUNTING\RECEIVABLES\MAY"
' Display the entire environment-string table.
PRINT "Your current environment settings are:"
PRINT
DO WHILE ENVIRON$(I%) <> ""
    PRINT ENVIRON$(I%)
    I% = I% + 1
LOOP
' Change the PATH back to original.
ENVIRON Path$ + OldPath$
' Verify the change.
PRINT
PRINT "Your PATH has been restored to:"
PRINT
PRINT ENVIRON$("PATH")
```

EOF Function

Action For an ISAM table, tests whether the current position is at the end of a table. For a non-ISAM file, tests for the end-of-file condition.

Syntax **EOF**(*filenumber%*)

Remarks The argument *filenumber%* is the number used in the **OPEN** statement to open the file or ISAM table.

The **EOF** function returns true (nonzero) if the end of a non-ISAM file has been reached or if the current position in an ISAM table is at the end of the table. The end of an ISAM table is the position immediately following the last record according to the current index. Use the **EOF** function with sequential files to test for the end of a file. This helps you avoid the `Input past end of file` error message.

When used with random-access or binary files, **EOF** returns true if the last executed **GET** statement was unable to read an entire record.

When you use **EOF** with a communications device, the definition of the end-of-file condition depends on the mode (ASCII or binary) in which you opened the device. In ASCII mode, **EOF** is false until you receive Ctrl+Z, after which it remains true until you close the device. In binary mode, **EOF** is true when the input queue is empty (**LOC**(*filenumber%*)=0), where *filenumber%* is the number of the device. It becomes false when the input queue is not empty.

Note **EOF** cannot be used with the BASIC devices SCRN, KYBD, CONS, LPT*n*, or PIPE.

See Also **BOF**, **LOC**, **LOF**, **OPEN**

Example See the **INPUT#** statement programming example, which uses the **EOF** function.

ERASE Statement

Action Reinitializes the elements of static arrays; deallocates dynamic arrays.

Syntax **ERASE** *arrayname* ⟦, *arrayname*⟧...

Remarks The *arrayname* arguments are the names of arrays to erase. It is important to know if this array is static or dynamic. **ERASE** sets the elements of an array as follows:

If type of array is:	ERASE sets array elements to:
Numeric static array	Zeros.
String static array	Null strings ("").
Array of records	Zeros—all elements of each record, including fixed-string elements.

Using **ERASE** on a dynamic array frees the memory used by the array. Before your program can refer to the dynamic array again, it must redeclare the array's dimensions with a **DIM** or **REDIM** statement. If you redeclare the array's dimensions with a **DIM** statement without first erasing it, BASIC generates the run-time error message `Array already dimensioned.` The **ERASE** statement is not required when dimensions are redeclared with **REDIM**.

See Also **CLEAR, DIM, REDIM**

Example The following example shows the use of **ERASE** with the **$DYNAMIC** and **$STATIC** metacommands:

```
REM $DYNAMIC
DIM A(100, 100)
' Deallocate array A.
ERASE A
' Redeclare dimensions for array A.
REDIM A(5, 5)

REM $STATIC
DIM B(50, 50)
' Set all elements of B equal to zero. (B still has the dimensions
' assigned by DIM.)
ERASE B
```

ERDEV, ERDEV$ Functions

Action Provide device-specific status information after an error.

Syntax ERDEV
ERDEV$

Remarks **ERDEV** is an integer function that returns an error code from the last device that generated a critical error. **ERDEV$** is a string function that returns the name of the device that generated the error. Because **ERDEV** and **ERDEV$** return meaningful information only after an error, they usually are used in error-handling routines specified by an **ON ERROR** statement.

ERDEV$ contains the 8-byte character device name if the error is on a character device, such as a printer, or the 2-byte block name (A:, B:, etc.) if the device is not a character device. It contains the 3-byte block name (COM) if the communications port experiences a timeout.

ERDEV is set by the critical error handler (interrupt 24H) when DOS detects a critical DOS call error. It also is set by a timeout error on the communications port and indicates which option in the **OPEN COM** statement (CD, CS, or DS) is experiencing the timeout.

ERDEV returns an integer value that contains information about the error. For block and character device errors, the low byte of **ERDEV** contains the DOS error code. For block devices only, the high byte contains device-attribute information.

For **COM** timeout errors, **ERDEV** returns a value corresponding to the source of the timeout. For more information, see the entry for the **OPEN COM** statement.

Assuming an **ERDEV** return value of x, the following program lines generate the DOS error code (low byte) and device attribute information (high byte).

```
DosErrCode = x AND &HFF              ' Low byte of ERDEV.
DevAttr = (x AND &HFF00) \ 256       ' High byte of ERDEV.
```

For more information about device-attribute words, see the *Microsoft MS-DOS Programmer's Reference*, or books such as *The Peter Norton Guide to the IBM PC* or *Advanced MS-DOS*.

Use **ERDEV** only in DOS.

Example

The following example prints the values of **ERDEV** and **ERDEV$** after the program generates an error attempting to open a file:

```
DEFINT A-Z
ON ERROR GOTO ErrorHandler    ' Indicate first line of error handler.
OPEN "A:JUNK.DAT" FOR INPUT AS #1    ' Attempt to open the file.
END

ErrorHandler:
   PRINT "ERDEV value is "; ERDEV
   PRINT "Device name is "; ERDEV$
   ON ERROR GOTO 0
```

Output

Running the program with drive A unlatched produces the following output (2 is the DOS error code for `Drive not ready`):

```
ERDEV value is  2
Device name is A:
```

ERR, ERL Functions

Action Return error status.

Syntax

ERR
ERL

Remarks After an error, the **ERR** function returns an integer that is the run-time code for the error. The **ERL** function returns an integer that is the line number where the error occurred, or the closest line number before the line where the error occurred. Because **ERR** and **ERL** return meaningful values only after an error, they usually are used in error-handling routines to determine the error and the corrective action.

The value returned by the **ERR** function can be directly set by using the **ERR** statement. Both the values returned by **ERR** and **ERL** can be set indirectly with the **ERROR** statement.

The **ERL** function returns only the line number, not line labels, located at or before the line producing the error. If your program has no line numbers, or there is no line number in the program before the point where the error occurred, **ERL** returns 0.

ERL values differ between programs compiled with BC and those run under QBX. In programs compiled with BC, **ERL** is set to the last numbered line in the source file preceding the line where the error occurred. When QBX detects an error in a procedure, it looks for a line number only within the immediate procedure. If the procedure doesn't contain a numbered line, QBX returns 0 for **ERL**.

You can make your programs run identically with both BC and QBX by always including a line number at the beginning of a procedure where an error might occur.

See Also **ERROR**; **ON ERROR**; **RESUME**; Table 4.1, "Run-Time Error Codes"

Example See the **ON ERROR** statement programming example, which uses the **ERR** function.

ERR Statement

Action Sets **ERR** to a specific value.

Syntax **ERR** = *n%*

Remarks The argument *n%* is an integer expression with a value between 1 and 255, inclusive, that specifies a run-time error code, or 0.

When running an application program, BASIC uses **ERR** to record whether or not a run-time error has occurred and what the error was. When a program starts running, **ERR** is 0; when and if a run-time error occurs, BASIC sets **ERR** to the error code for that error.

You may want to use the **ERR** statement to set **ERR** to a non-zero value to communicate error information between procedures. For example, you might use one of the run-time codes not used by BASIC as an application-specific error code. See Table 4.1, "Run-Time Error Codes," for a list of the run-time error codes that BASIC uses; they are a subset of the integers between 1 and 255, inclusive.

Besides the **ERR** statement, the following BASIC statements set **ERR** whenever they execute:

- Any form of the **RESUME** statement sets **ERR** to 0.
- **EXIT SUB**, **EXIT FUNCTION**, or **EXIT DEF** sets **ERR** to 0 if executed within a procedure-level error handler.
- All uses of the **ON ERROR** or **ON LOCAL ERROR** statements syntax set **ERR** to 0.
- The **ERROR** statement can be used to set **ERR** to any value as part of simulating any run-time error.

See Also **ERR**, **ERL**; **ERROR**; Table 4.1, "Run-Time Error Codes"

Example See the **ON ERROR** statement programming example, which uses the **ERR** statement.

ERROR Statement

Action Simulates the occurrence of a BASIC error or a user-defined error.

Syntax **ERROR** *integerexpression%*

Remarks The argument *integerexpression%* represents the error code. It must be between 1 and 255, inclusive. If *integerexpression%* is an error code already used by BASIC, the **ERROR** statement simulates the occurrence of that error.

To define your own error code, use a value that is greater than any used by the standard BASIC error codes. (Start at 255 and work down to avoid compromising compatibility with future Microsoft BASIC error codes.) In general, the error codes used by BASIC are between 1 and 100 (although not all these are used).

If an error statement is executed when no error-handling routine is enabled, BASIC generates an error message and stops program execution. If the **ERROR** statement specified an error code that is not used by BASIC, the message `Unprintable error` is generated.

See Also **ERR, ERL; ON ERROR; RESUME**

Example See the **ON ERROR** statement programming example, which uses the **ERROR** statement.

EVENT Statements

Action Enable or disable trapping of events.

Syntax **EVENT ON**
EVENT OFF

Remarks **EVENT ON** enables trapping of events until BASIC encounters the next **EVENT OFF** statement.

EVENT OFF disables trapping of events until BASIC encounters the next **EVENT ON** statement.

The **EVENT** statements affect compiled code generation. When **EVENT OFF** is in effect, the compiler will not generate any event-checking code between statements. **EVENT OFF** is equivalent to compiling a program without /V or /W. **EVENT OFF** can be used to create small and fast code that still supports event trapping.

If your program contains event-handling statements and you are compiling from the BC command line, use the BC /W or /V option. (The /W option checks for events at every label or line number; the /V option checks at every statement.) If you do not use these options and your program contains event traps, BASIC generates the error message `ON event without /V or /W on command line.`

EVENT OFF and **EVENT ON** can be used to bracket sections of code where events do not need to be detected and trapped, and fast performance is required. Events still could be detected and tracked if a subroutine using **EVENT ON** were called from a section of **EVENT OFF** code.

If an event occurs while **EVENT OFF** is in effect, this event still is remembered and then is handled as soon as an **EVENT ON** block is entered.

See Also **ON** *event*

Example See the **STRIG** statements programming example, which uses the **EVENT** statements.

EXIT Statement

Action Exits a **DEF FN** function, a **DO...LOOP** or **FOR...NEXT** loop, or a **FUNCTION** or **SUB** procedure.

Syntax **EXIT** {**DEF** | **DO** | **FOR** | **FUNCTION** | **SUB**}

Remarks There are a number of ways to use the **EXIT** statement, as described in the following list:

Statement	*Description*
EXIT DEF	Causes an immediate exit from the executing **DEF FN** function. Program execution continues where the **DEF FN** function was invoked.
EXIT DO	Provides an alternative exit from a **DO...LOOP** statement. Can be used only inside a **DO...LOOP** statement; **EXIT DO** transfers control to the statement following the **LOOP** statement. When used within nested **DO...LOOP** statements, transfers out of the immediately enclosing loop.
EXIT FOR	Provides another way to exit a **FOR...NEXT** loop. May appear only in a **FOR...NEXT** loop; transfers control to the statement following the **NEXT** statement. When used within nested **FOR...NEXT** loops, transfers out of the immediately enclosing loop.
EXIT FUNCTION	Causes an immediate exit from a **FUNCTION** procedure. Program execution continues where the procedure was invoked. Can be used only in a **FUNCTION** procedure.
EXIT SUB	Immediately exits a **SUB** procedure. Program execution continues with the statement after the **CALL** statement. Can be used only in a **SUB** procedure.

None of the **EXIT** statements defines the end of the structure in which it is used. **EXIT** statements provide only an alternative exit from the structure.

See Also **DEF FN, DO...LOOP, FOR...NEXT, FUNCTION, SUB**

Example The following example demonstrates the use of a variety of **EXIT** statements. A loop is continuously executed until a key is pressed. Once a key is pressed, the next **EXIT** statement that executes will cause the program to end.

```
DECLARE SUB ExitDemo ()
CLS
DO
   PRINT : PRINT "Entering/Re-entering ExitDemo"
   ExitDemo
   SLEEP 1
LOOP WHILE INKEY$ = ""
PRINT "Exiting EXIT statement programming example."
END

SUB ExitDemo
   DO
      FOR I% = 1 TO 1000
         Num% = INT(RND * 100)
         SELECT CASE Num%
            CASE 7
               PRINT "Exiting FOR...NEXT loop in ExitDemo SUB"
               EXIT FOR
            CASE 29
               PRINT "Exiting DO...LOOP in ExitDemo SUB"
               EXIT DO
            CASE 54
               PRINT "Exiting ExitDemo SUB"
               EXIT SUB
            CASE ELSE
         END SELECT
      NEXT I%
   LOOP
END SUB
```

EXP Function

Action Calculates the exponential function.

Syntax **EXP**(*x*)

Remarks The **EXP** function returns *e* (the base of natural logarithms) to the power of *x*.

The exponent *x* must be less than or equal to 88.02969 when you are using single-precision values and less than or equal to 709.782712893 when you are using double-precision values. If you use a value of *x* that isn't within those limits, BASIC generates the error message `Overflow.`

EXP is calculated in single precision if *x* is an integer or single-precision value. If you use any other numeric data type, **EXP** is calculated in double precision.

See Also **LOG**

Example The following example uses the **EXP** function to calculate the growth of a bacterial colony over a 15-day period. The program prompts you for an initial population and the rate of growth.

```
CLS    ' Clear screen.
INPUT "Initial bacterial population"; Colony0
INPUT "Growth rate per day as a percentage of population"; Rate
R = Rate / 100 : Form$ = "##  ###,###,###,###"
PRINT : PRINT "Day        Population"
FOR T = 0 TO 15 STEP 5
    PRINT USING Form$; T, Colony0 * EXP(R * T)
NEXT
```

Output

```
Initial bacterial population? 10000
Growth rate per day as a percentage of population? 10

Day        Population
 0           10,000
 5           16,487
10           27,183
15           44,817
```

FIELD Statement

Action Allocates space for variables in a random-access file buffer.

Syntax **FIELD** [[#]]*filenumber%*, *fieldwidth%* **AS** *stringvariable$* [[, *fieldwidth%* **AS** *stringvariable$*]]...

Remarks The **FIELD** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
fieldwidth%	The number of characters in a field.
stringvariable$	The string variable that contains the data read from a record, or data that is used in an assignment when information is written to a record.

The total number of bytes that you allocate in a **FIELD** statement must not exceed the record length you specified when opening the file. Otherwise, BASIC generates an error message that reads `FIELD overflow`. (The default record length is 128 bytes.)

Any number of **FIELD** statements can be executed for the same file. All **FIELD** statements that have been executed remain in effect at the same time.

All field definitions for a file are removed when the file is closed; that is, all strings defined as fields associated with the file are set to null.

Note Do not use a variable name defined as a field in an **INPUT** or assignment statement if you want the variable to remain a field. Once a variable name is a field, it points to the correct place in the random-access file buffer. If a subsequent **INPUT** or assignment statement with that variable name is executed, the variable's pointer no longer refers to the random-access record buffer, but to string space.

BASIC's record variables and extended **OPEN** statement syntax provide a more convenient way to use random-access files. See Chapter 3, "File and Device I/O" in the *Programmer's Guide* for an extended discussion of using record variables for file I/O.

BASICA When a random-access file is closed with a **CLOSE** or **RESET** statement in a compiled program, all variables that are fields associated with that file are reset to null strings. When a random-access file is closed in a BASICA program, variables that are fields retain the last value assigned to them by a **GET** statement.

See Also **GET** (File I/O), **LSET**, **PUT** (File I/O), **RSET**

Example

The example below illustrates a random-access file buffer with multiple definitions.

In the first **FIELD** statement, the 67-byte buffer is broken up into five separate variables for name, address, city, state, and zip code. In the second **FIELD** statement, the same buffer is assigned entirely to one variable, `Plist$`.

The remainder of this example checks to see if `Zip$`, which contains the zip code, falls within a certain range; if it does, the complete address string is printed.

```
TYPE Buffer
   FuName AS STRING * 25
   Addr   AS STRING * 25
   City   AS STRING * 10
   State  AS STRING * 2
   Zip    AS STRING * 5
END TYPE
DIM RecBuffer AS Buffer

' This part of the program creates a random-access file for use by the
' second part of the program, which demonstrates the FIELD statement.
OPEN "MAILLIST.DAT" FOR RANDOM AS #1 LEN = LEN(RecBuffer)
CLS
RESTORE
READ FuName$, Addr$, City$, State$, Zip$
I = 0
DO WHILE UCASE$(FuName$) <> "END"
   I = I + 1
   RecBuffer.FuName = FuName$
   RecBuffer.Addr = Addr$
   RecBuffer.City = City$
   RecBuffer.State = State$
   RecBuffer.Zip = Zip$
   PUT #1, I, RecBuffer
   READ FuName$, Addr$, City$, State$, Zip$
   IF FuName$ = "END" THEN EXIT DO
LOOP
CLOSE #1
DATA "Bob Hartzell","1200 Liberty St.","Bow","WA","98232"
DATA "Alice Provan","123 B St.","Bellevue","WA","98005"
DATA "Alex Landow","14900 123rd","Bothell","WA","98011"
DATA "Walt Riley","33 Minnow Lake Road","Lyman","WA","98263"
DATA "Georgette Gump","400 15th W.","Bellevue","WA","98007"
DATA "END",0,0,0,0,0
```

```
' Define field and record lengths with constants.
CONST FU = 25, AD = 25, CT = 10, ST = 2, ZP = 5
CONST RECLEN = FU + AD + CT + ST + ZP

OPEN "MAILLIST.DAT" FOR RANDOM AS #1 LEN = RECLEN
FIELD #1, FU AS FuName$, AD AS Addr$, CT AS City$, ST AS State$, ZP AS Zip$
FIELD #1, RECLEN AS Plist$
GET #1, 1
' Read the file, looking for zip codes in the range 98000 to 98015.
DO WHILE NOT EOF(1)
   Zcheck$ = Zip$
   IF (Zcheck$ >= "98000" AND Zcheck$ <= "98015") THEN
      Info$ = Plist$
      PRINT LEFT$(Info$, 25)
      PRINT MID$(Info$, 26, 25)
      PRINT RIGHT$(Info$, 17)
      PRINT
   END IF
   GET #1
LOOP
CLOSE #1
```

FILEATTR Function

Action Returns information about an open file or ISAM table.

Syntax **FILEATTR**(*filenumber%*,*attribute%*)

Remarks **FILEATTR** returns information about an open file or ISAM table, such as the DOS file handle and whether the file was opened for input, output, random, append, or ISAM binary mode.

The **FILEATTR** function uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file or ISAM table. You can use a numeric expression as long as BASIC evaluates it to the number of an open file, device, or ISAM table.
attribute%	Indicates the type of information to return about the file or table. When *attribute%* is 1, **FILEATTR** returns a value indicating the file's mode (see below). When *attribute%* is 2 and *filenumber%* indicates a file, **FILEATTR** returns the DOS handle for the file. When *attribute%* is 2 and *filenumber%* indicates an ISAM table, **FILEATTR** returns 0.

The following table lists the return values and corresponding file modes when *attribute%* is 1:

Return value	Mode
1	Input
2	Output
4	Random access
8	Append
32	Binary
64	ISAM

See Also **OPEN** (File I/O)

Example The following example opens a file and displays its DOS file handle and mode:

```
OPEN "tempfile.dat" FOR APPEND AS #1
PRINT "Handle: "; FILEATTR(1,2); "      Mode: "; FILEATTR(1,1)
```

Output

```
Handle:  5      Mode:  8
```

FILES Statement

Action Prints the names of files residing on the specified disk.

Syntax **FILES** ⟦*filespec$*⟧

Remarks The argument *filespec$* is a string variable or constant that includes either a filename or a path, and an optional device name.

If no argument is specified, the **FILES** statement lists all the files in the current directory. You can use the DOS wildcard characters—question marks (?) or asterisks (*). A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at that position.

If you use *filespec$* without an explicit path, the current directory is the default.

Examples The statements below illustrate the use of **FILES**.

Note that execution halts if you try to run this example without a disk in drive B or if the specified files cannot be found.

```
FILES                ' Show all files on the current directory.
FILES "*.BAS"        ' Show all files with the extension .BAS.
FILES "B:*.*"        ' Show all files in current directory of drive B.
FILES "B:"           ' Equivalent to "B:*.*".
FILES "TEST?.BAS"    ' Show all five-letter files whose names start
                     ' with "TEST" and end with the .BAS extension.
FILES "SALES\"       ' If SALES is a directory, this statement
                     ' displays all files in SALES.
```

FIX Function

Action Returns the truncated integer part of a numeric expression.

Syntax **FIX**(*x#*)

Remarks The argument *x#* is a numeric expression. **FIX**(*x#*) is equivalent to SGN(*x#*)*INT(ABS(*x#*)). The difference between **FIX** and **INT** is that for negative *x#*, **FIX** returns the first negative integer greater than *x#*, while **INT** returns the first negative integer less than *x#*.

See Also **CINT, INT**

Example The following statements illustrate the differences between **INT** and **FIX**:

```
PRINT INT(-99.8)
PRINT FIX(-99.8)
PRINT INT(-99.2)
PRINT FIX(-99.2)
```

Output

```
-100
-99
-100
-99
```

FOR...NEXT Statement

Action Repeats a group of instructions a specified number of times.

Syntax

FOR *counter* = *start* **TO** *end* ⟦**STEP** *increment*⟧
  ⟦*statementblock*⟧
  ⟦**EXIT FOR**⟧
  ⟦*statementblock*⟧
NEXT ⟦*counter* ⟦, *counter*⟧...⟧

Remarks The **FOR** statement takes the following arguments:

Argument	Description
counter	A numeric variable used as the loop counter. The variable cannot be an array element or a record element.
start	The initial value of the counter.
end	The final value of the counter.
increment	The amount the counter is changed each time through the loop. If you do not specify **STEP**, *increment* defaults to one.

A **FOR...NEXT** loop executes only if *start* and *end* are consistent with *increment*. If *end* is greater than *start*, *increment* must be positive. If *end* is less than *start*, *increment* must be negative. BASIC checks this at run time by comparing the sign of (*end* – *start*) with the sign of **STEP**. If both have the same sign, the **FOR...NEXT** loop is entered. If not, the entire loop is skipped.

Within the **FOR...NEXT** loop, the program lines following the **FOR** statement are executed until the **NEXT** statement is encountered. Then *counter* is changed by the amount specified by **STEP**, and compared with the final value, *end*.

If *start* and *end* have the same value, the loop executes once, regardless of the value of **STEP**. Otherwise, **STEP** value controls the loop as follows:

STEP value	Loop execution
Positive	If *counter* is less than or equal to *end*, control returns to the statement after the **FOR** statement and the process repeats. If *counter* is greater than *end*, the loop is exited; execution continues with the statement following the **NEXT** statement.
Negative	The loop repeats until *counter* is less than *end*.
Zero	The loop repeats indefinitely.

Avoid changing the value of *counter* within the loop. Changing the loop counter is poor programming practice; it can make the program more difficult to read and debug.

You can nest **FOR...NEXT** loops; that is, you can place a **FOR...NEXT** loop within another **FOR...NEXT** loop. To ensure that nested loops work properly, give each loop a unique variable name as its counter. The **NEXT** statement for the inside loop must appear before the **NEXT** statement for the outside loop. The following construction is correct:

```
FOR I = 1 TO 10
   FOR J = 1 TO 10
      FOR K = 1 TO 10
      .
      .
      .
      NEXT K
   NEXT J
NEXT I
```

A **NEXT** statement with the form `NEXT K, J, I` is equivalent to the following sequence of statements:

```
NEXT K
NEXT J
NEXT I
```

The **EXIT FOR** statement is a convenient alternative exit from **FOR...NEXT** loops. See the entry for **EXIT**.

Note

If you omit the variable in a **NEXT** statement, the **NEXT** statement matches the most recent **FOR** statement. If a **NEXT** statement is encountered before its corresponding **FOR** statement, BASIC generates the error message `NEXT without FOR`.

BASICA

Unlike BASICA, BASIC supports double-precision control values (*start*, *end*, and *counter*) in its **FOR...NEXT** loops. However, if the control values fall within the range for integers, you should use integer control values for maximum speed.

Example

The following example prints the first 11 columns of Pascal's Triangle, in which each number is the sum of the number immediately above it and the number immediately below it in the preceding column:

```
DEFINT A-Z
CLS                             ' Clear screen.
CONST MAXCOL = 11
DIM A(MAXCOL, MAXCOL)
PRINT "Pascal's Triangle"
FOR M = 1 TO MAXCOL
    A(M, 1) = 1: A(M, M) = 1   ' Top and bottom of each column is 1.
NEXT
```

```
FOR M = 3 TO MAXCOL
   FOR N = 2 TO M - 1
      A(M, N) = A(M - 1, N - 1) + A(M - 1, N)
   NEXT
NEXT
Startrow = 13                  ' Go to the middle of the screen.
FOR M = 1 TO MAXCOL
   Col = 6 * M
   Row = Startrow
   FOR N = 1 TO M
      LOCATE Row, Col: PRINT A(M, N)
      Row = Row + 2            ' Go down 2 rows to print next number.
   NEXT
   PRINT
   Startrow = Startrow - 1     ' Next column starts 1 row above
NEXT                           ' preceding column.
```

Output

```
Pascal's Triangle
                                                              1
                                                      1
                                                1            10
                                          1           9
                                    1           8            45
                              1           7           36
                        1           6           28          120
                  1           5           21          84
            1           4           15          56          210
      1           3           10          35          126
1           2           6           20          70          252
      1           3           10          35          126
            1           4           15          56          210
                  1           5           21          84
                        1           6           28          120
                              1           7           36
                                    1           8            45
                                          1           9
                                                1            10
                                                      1
                                                              1
```

FRE Function

Action Returns a value representing the amount of available memory, depending upon a given argument.

Syntax 1 **FRE**(*numeric-expression%*)

Syntax 2 **FRE**(*stringexpression$*)

Remarks The argument for **FRE** can be either a string or numeric expression. The argument *stringexpression$* can be a string literal or a string variable.

The values returned for the different types of arguments depend on whether you are using near strings or far strings, as described in the following table:

Function	Value returned *if using near strings*	Value returned *if using far strings*
FRE(a$)	Remaining space in DGROUP (in bytes)	Remaining space in a$'s segment (in bytes)
FRE("string literal")	Remaining space in DGROUP (in bytes)	Remaining space for temporary strings (in bytes)
FRE(0)	Remaining space in DGROUP (in bytes)	Error message: `Illegal function call`
FRE(–1)	For DOS, remaining space (in bytes) in far memory; for OS/2, long integer 2147483647	For DOS, remaining space (in bytes) in far memory; for OS/2, long integer 2147483647
FRE(–2)	Remaining stack space (in bytes)	Remaining stack space (in bytes)
FRE(-3)	Remaining space in expanded memory (in kilobytes)	Remaining space in expanded memory (in kilobytes)
FRE(any other no.)	Error message: `Illegal function call`	Error message: `Illegal function call`

`FRE(-2)` returns the amount of stack space that was never used by the program. For example, suppose a program uses a lot of stack space in a recursive procedure and then returns to the main level to execute a `FRE(-2)`. The value returned will be the amount of stack space that was never used by the program. Any space used by the recursive procedure is considered "used" So `FRE(-2)` enables you to find out at the completion of your program what the worst-case stack usage was for that program.

Using FRE(-3) and Expanded Memory

If expanded memory is not available, `FRE(-3)` generates the error message `Feature unavailable.`

If you start QBX without the /E:*n* switch, you will be using all of the expanded memory available on your computer. If you start QBX with the /E:*n* switch, you can limit the amount of expanded memory available to your program. For example, if your computer has 5 megabytes of expanded memory but you want to use only 4 megabytes of expanded memory, you would start QBX with this command line:

```
QBX /E:4096
```

If you have specified 4 megabytes of expanded memory as shown above, and then you use 2 megabytes of expanded memory in your program, `FRE(-3)` will return 2048.

If you are using BC to run a program with expanded memory overlays, the value returned by FRE(-3) is based on the total amount of expanded memory available. (When using overlays, you cannot limit the amount of expanded memory used by your program.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started.*

Note `FRE(-2)` returns meaningful values only when a program is executing. Values returned by `FRE(-2)` are not accurate when the function is called from the Immediate window, during program tracing, or when monitoring a variable.

Example The following example demonstrates use of the **FRE** function to report the availability of memory resources. It also uses the **STACK** function to allocate stack space.

```
DECLARE SUB MakeSubString ()
DECLARE SUB Printit (A$)
DECLARE SUB Recursive (n!)
PRINT "Remaining space to begin with is:"
CALL Printit(A$)
String1$ = STRING$(32767, 90) ' Create a 32K string.
PRINT " The remaining space after creating a 32K far string is:"
CALL Printit(String1$)
MakeSubString   ' Make a substring and give report.
PRINT "The remaining space after returning from a SUB is:"
CALL Printit(String2$)
n = 50          ' Do 50 recursive calls to a SUB procedure.
CALL Recursive(n)
STACK (2048)
PRINT "After allocating 2K for the stack, the space is:"
CALL Printit(A$)
n = 50          ' Do another 50 recursive calls to a SUB procedure.
CALL Recursive(n)
```

```
'Dimension a 1001-element dynamic array in EMS.
REDIM A%(999)
PRINT "After dimensioning a 1000-element integer array, the space is:"
CALL Printit(A$)
PRINT "After dimensioning a second 1000-element integer array:"
CALL Printit(A$)

PRINT "Stack reset to default value: "; STACK

SUB MakeSubString
SHARED String1$
    String2$ = STRING$(32767, 90)
    PRINT "The space remaining after creating a 32K sub string is:"
    CALL Printit(String1$)
END SUB

SUB Printit (A$)
    PRINT : PRINT "Far Memory"; TAB(15); "Stack"; TAB(25);
    PRINT "String Segment";TAB(45);"Temporary Segment";TAB(65);"EMS Memory"
    PRINT FRE(-1); TAB(15); FRE(-2); TAB(25); FRE(A$); TAB(45); FRE("")
    PRINT TAB(65); FRE(-3); "K"
END SUB

SUB Recursive (n)
SHARED String1$
    n = n - 1
    IF n = 0 THEN
        PRINT "The remaining space after 50 recursive calls is:"
        CALL Printit(String1$)
        EXIT SUB
    ELSE
        CALL Recursive(n)
    END IF
END SUB
```

FREEFILE Function

Action Returns the next valid unused file number.

Syntax **FREEFILE**

Remarks Use **FREEFILE** when you need to supply a file number and you want to ensure that the file number is not already in use.

See Also **OPEN** (File I/O)

Example The following example uses **FREEFILE** to obtain the next available file number for opening a sequential file. **OPEN**, **CLOSE**, and **KILL** also are used.

```
DIM Filenum AS INTEGER
CLS
INPUT "Enter filename: ", filename$
Filenum% = FREEFILE
OPEN filename$ FOR OUTPUT AS Filenum%
PRINT
PRINT UCASE$(filename$); " opened for output as File #"; Filenum%
' Put something out to the file.
PRINT #Filenum%, "The quick brown fox jumped over the lazy dog."
' Close the file just opened.
CLOSE Filenum%
Filenum% = FREEFILE
OPEN filename$ FOR INPUT AS Filenum%
PRINT : PRINT UCASE$(filename$); " has been reopened for input."
LINE INPUT #Filenum%, L$
PRINT : PRINT "The contents of the file are:"; L$
CLOSE Filenum%
' Remove the file from disk.
KILL filename$
```

FUNCTION Statement

Action Declares the name, the parameters, and the return type of a **FUNCTION** procedure.

Syntax

FUNCTION *name* ⟦(*parameterlist*)⟧ ⟦**STATIC**⟧
  ⟦*statementblock*⟧
  name = *expression*
  ⟦*statementblock*⟧
  ⟦**EXIT FUNCTION**⟧
  ⟦*statementblock*⟧
END FUNCTION

Remarks The following list describes the parts of the **FUNCTION** statement:

Part	Description
name	The name of the function. **FUNCTION** procedure names follow the same rules as are used for naming BASIC variables and can include a type-declaration character (%, &, !, #, @, or $). Note that the type of the name determines the data type the function returns. For example, to create a function that returns a string, you would include a dollar sign in the name or give it a name defined as a string name by a **DEFSTR** statement.
parameterlist	The list of variables, representing parameters, that will be passed to the **FUNCTION** procedure when it is called. Multiple variables are separated by commas. Parameters are passed by reference, so any change to a parameter's value inside the function changes its value in the calling program.
STATIC	Indicates that the function's local variables are to be saved between calls. Without **STATIC**, the local variables are allocated each time the function is invoked, and the variables' values are lost when the function returns to the calling program. The **STATIC** attribute does not affect variables that are used in a **FUNCTION** procedure but declared outside the procedure in **DIM** or **COMMON** statements using the **SHARED** statement.
expression	The return value of the function. A **FUNCTION** procedure returns a value by assigning a value to the function name. If no value is assigned to the **FUNCTION** name, the procedure returns a default value: a numeric function returns zero, and a string function returns the null string ("").

EXIT FUNCTION	Causes an immediate exit from a **FUNCTION** procedure. Program execution continues where the procedure was invoked. For more information, see the entry for the **EXIT** statement.

The argument *parameterlist* has the following syntax:

variable⟦( )⟧ ⟦**AS** *type*⟧ ⟦, *variable*⟦( )⟧ ⟦**AS** *type*⟧⟧...

The following list describes the parts of *parameterlist* :

Argument	Description
variable	A BASIC variable name. Previous versions of BASIC required the number of dimensions in parentheses after an array name. In the current version of BASIC, the number of dimensions is not required.
AS *type*	The type of the variable: **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, **CURRENCY**, or a user-defined type. You cannot use a fixed-length string, or an array of fixed-length strings, as a parameter. However, you can use a simple fixed-length string as an argument in a **CALL** statement; BASIC converts a simple fixed-length string argument to a variable-length string argument before passing the string to a **FUNCTION** procedure.

Earlier versions of BASIC required the number of dimensions in parentheses after an array name. The number of dimensions is no longer required. Only the parentheses are required to indicate the parameter is an array. For example, the following statement indicates that both `Keywords$` and `KeywordTypes` are arrays:

```
FUNCTION ParseLine(Keywords$(),KeywordTypes())
```

A **FUNCTION** procedure is like a **SUB** procedure: it can accept parameters, perform a series of statements, and change the values of its parameters. Unlike a **SUB** procedure, a **FUNCTION** procedure is used in an expression in the same manner as a BASIC intrinsic function.

Like **SUB** procedures, **FUNCTION** procedures use local variables. Any variable referred to within the body of the function is local to the **FUNCTION** procedure unless one of the following conditions is true:

- The variable is declared within the function as a shared variable with a **SHARED** statement.
- The variable appears in a **DIM** or **COMMON** statement with the **SHARED** attribute at the module level.

To return a value from a function, assign the value to the function name. For example, in a function named `BinarySearch`, you might assign the value of the constant `FALSE` to the name to indicate the value was not found:

```
FUNCTION BinarySearch(...)
CONST FALSE=0
.
.
.
   ' Value not found. Return a value of FALSE.
   IF Lower>Upper THEN
      BinarySearch=FALSE
      EXIT FUNCTION
   END IF
.
.
.
END FUNCTION
```

Using the **STATIC** attribute increases execution speed slightly. **STATIC** usually is not used with recursive **FUNCTION** procedures.

Note

Avoid using I/O statements in a **FUNCTION** procedure called from an I/O statement; they can cause unpredictable results. Also, because BASIC may rearrange arithmetic expressions to attain greater efficiency, avoid using **FUNCTION** procedures that change program variables in arithmetic expressions.

FUNCTION procedures are recursive—they can call themselves to perform a given task. See the example for more information.

See Also

DECLARE (BASIC), **DEF FN**, **STATIC** Statement, **SUB**

Example

The following example uses a recursive **FUNCTION** procedure (a procedure that calls itself) to find, and return as an integer, the length of a string:

```
DECLARE FUNCTION StrgLng% (X$).
INPUT "Enter a string: "; InString$
PRINT "The string length is"; StrgLng%(InString$)

FUNCTION StrgLng% (X$)
   IF X$ = "" THEN
      StrLng% = 0   ' The length of a null string is zero.
   ELSE
      StrgLng% = 1 + StrgLng%(MID$(X$, 2))   ' Make a recursive call.
   END IF
END FUNCTION
```

GET Statement (File I/O)

Action Reads from a disk file into a random-access buffer or variable.

Syntax **GET** [[#]]*filenumber%*[[,[[*recordnumber%*]][[,*variable*]]]]

Remarks Do not use **GET** on ISAM files.

The **GET** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
recordnumber%	For random-access files, the number of the record to be read. For binary-mode files, the byte position where reading starts. The first record or byte position in a file is 1. If you omit *recordnumber%*, the next record or byte (the one after the last **GET** or **PUT** statement, or the one pointed to by the last **SEEK**) is read into the buffer. The largest possible record number is $2^{31}-1$, or 2,147,483,647.

variable	The variable used to receive input from the file. If you specify a variable, you do not need to use **CVI**, **CVL**, **CVS**, **CVD**, or **CVC** to convert record fields to numbers. You cannot use a **FIELD** statement with the file if you use the *variable* argument.
	
	For random-access files, you can use any variable as long as the length of the variable is less than or equal to the length of the record. Usually, a record variable defined to match the fields in a data record is used.
	
	For binary-mode files, you can use any variable. The **GET** statement reads as many bytes as there are in the variable.
	
	When you use a variable-length string variable, the statement reads as many bytes as there are characters in the string's value. For example, the following two statements read 10 bytes from file number 1:
	
	`VarStrings$=STRING$ (10, " ")`
	`GET #1,,VarString$`
	
	See the examples for more information about using variables rather than **FIELD** statements for random-access files.

A record cannot be longer than 32,767 bytes.

You can omit *recordnumber%*, *variable*, or both. If you omit *recordnumber%* but include a variable, you must still include the commas:

```
GET #4,,FileBuffer
```

If you omit both arguments, do not include the commas:

```
GET #4
```

GET and **PUT** statements allow fixed-length input and output for BASIC communication files. Use **GET** carefully, because if there is a communication failure, **GET** waits indefinitely.

Note When you use **GET** with the **FIELD** statement, you can use **INPUT #** or **LINE INPUT #** after a **GET** statement to read characters from the random-access file buffer. You can use the **EOF** function after a **GET** statement to see if the **GET** operation went beyond the end of the file.

See Also **CVI, CVL, CVS, CVD, CVC**; **LSET**; **MKI$, MKL$, MKS$, MKD$, MKC$**; **OPEN** (File I/O); **PUT** (File I/O)

Example The following example creates a random-access test file that contains five names and corresponding test scores. It then displays the contents of the file using the **GET** statement.

```
' Define record fields.
TYPE TestRecord
    NameField  AS STRING * 20
    ScoreField AS SINGLE
END TYPE
' Define a variable of the user type.
DIM Rec AS TestRecord
DIM I AS LONG
```

```
' This part of the program creates a random-access file to be used by
' the second part of the program, which demonstrates the GET statement.
OPEN "TESTDAT2.DAT" FOR RANDOM AS #1 LEN = LEN(Rec)
CLS
RESTORE
READ NameField$, ScoreField
I = 0
DO WHILE NameField$ <> "END"
    I = I + 1
    Rec.NameField = NameField$
    Rec.ScoreField = ScoreField
    PUT #1, I, Rec
    READ NameField$, ScoreField
LOOP
CLOSE #1
DATA "John Simmons", 100
DATA "Allie Simpson", 95
DATA "Tom Tucker", 72
DATA "Walt Wagner", 90
DATA "Mel Zucker", 92
DATA "END", 0

' Open the test data file.
DIM FileBuffer AS TestRecord
DIM Max AS LONG
OPEN "TESTDAT2.DAT" FOR RANDOM AS #1 LEN = LEN(FileBuffer)
'Calculate number of records in the file.
Max = LOF(1) \ LEN(FileBuffer)

' Read and print contents of each record.
FOR I = 1 TO Max
    GET #1, I, FileBuffer
    PRINT FileBuffer.NameField, FileBuffer.ScoreField
NEXT I
CLOSE #1
' Remove file from disk.
KILL "TESTDAT2.DAT"
END
```

GET Statement (Graphics)

Action Stores graphic images from the screen.

Syntax **GET** ⟦**STEP**⟧(*x1!*, *y1!*) – ⟦**STEP**⟧(*x2!*, *y2!*), *arrayname*⟦(*indexes%*)⟧

Remarks The list below describes the parts of the **GET** statement:

Part	Description
(*x1!*, *y1!*),(*x2!*, *y2!*)	Coordinates that mark a rectangular area on the screen; an image within this rectangle will be stored. The *x1!*, *y1!*, *x2!*, and *y2!* are numeric expressions. The coordinates (*x1!*, *y1!*) and (*x2!*, *y2!*) are the coordinates of diagonally opposite corners of the rectangle.
STEP	Keyword indicating that coordinates are relative to the most recently plotted point. For example, if the last point plotted were (10,10), the actual coordinates referred to by `STEP (5,10)` would be (5+10,10+10) or (15,20). If the second coordinate pair in a **GET** statement has a **STEP** argument, it is relative to the first coordinate pair in the statement.
arrayname	Name assigned to the array that holds the image. This array can be of any numeric type; its dimensions must be large enough to hold the entire image.
indexes%	Numeric constants or variables indicating the element of the array where the saved image starts.

The **GET** statement transfers a screen image into the array specified by *arrayname*. The **PUT** statement, associated with **GET**, transfers the image stored in the array onto the screen.

The following formula gives the required size of the array in bytes:

size = 4 + INT(((*x2!* – *x1!* + 1) * (*bits-per-pixel-per-plane*) + 7)/8) * *planes* * ((*y2!* – *y1!*) + 1)

The *bits-per-pixel-per-plane* and *planes* values depend on the screen mode set by the **SCREEN** statement. The following table shows the number of bits-per-pixel-per-plane and the number of planes for each screen mode:

Screen mode	Bits per pixel per plane	Planes
1	2	1
2	1	1
7	1	4
8	1	4
9 (64K of EGA memory)	1	2
9 (> 64K of EGA memory)	1	4
10	1	2
11	1	1
12	1	4
13	8	1

The bytes per element of an array are as follows:

- Two bytes for an integer array element.
- Four bytes for a long-integer array element.
- Four bytes for a single-precision array element.
- Eight bytes for a double-precision array element.
- Eight bytes for a currency array element.

For example, suppose you wanted to use the **GET** statement to store an image in high resolution (`SCREEN 2`). If the coordinates of the upper-left corner of the image are (0,0), and the coordinates of the lower-right corner are (32,32), then the required size of the array in bytes is 4 + INT((33 * 1 + 7)/8) * 1 * (33), or 169. This means an integer array with 85 elements would be large enough to hold the image.

Unless the array type is integer or long, the contents of an array after a **GET** operation appear meaningless when inspected directly. Examining or manipulating noninteger arrays containing graphics images may cause run-time errors.

GET and **PUT** statements operating on the same image should be executed in matching screen modes. These modes can be either the same screen mode or any screen modes with the same values for planes and bits-per-pixel-per-plane.

One of the most useful things that can be done with **GET** and **PUT** statements is animation. See Chapter 5, "Graphics" in the *Programmer's Guide* for a discussion of animation.

See Also **PUT** (Graphics)

Example See the **BSAVE** statement programming example, which uses the **GET** statement.

GETINDEX$ Function

Action Returns the name of the current index for an ISAM table.

Syntax **GETINDEX$** (*filenumber%*)

Remarks The *filenumber%* is the number used in the **OPEN** statement to open the table.

GETINDEX$ takes the number of an open ISAM table and returns the name of the current index. If the current index is the NULL index, the value returned is a null string. The NULL index represents the order in which records were added to the file.

See Also **CREATEINDEX**, **DELETEINDEX**, **OPEN** (File I/O), **SETINDEX**

Example See the **CREATEINDEX** statement programming example, which uses the **GETINDEX$** function.

GOSUB...RETURN Statements

Action Branch to, and return from, a subroutine.

Syntax

GOSUB {*linelabel1* | *linenumber1*}
.
.
.
RETURN [[{*linelabel2* | *linenumber2*}]]

Remarks The **GOSUB...RETURN** statements use the following arguments:

Argument	Description
linelabel1, linenumber1	The line label or line number that is the first line of the subroutine.
linelabel2, linenumber2	The line label or line number where the subroutine returns.

Note BASIC's **SUB** and **FUNCTION** procedures provide a better-structured alternative to **GOSUB...RETURN** subroutines.

In addition to **RETURN** with no argument, BASIC supports **RETURN** with a line label or line number. This allows a return from a subroutine to the statement having the specified line label or number, instead of returning to the statement after the **GOSUB** statement. Use this line-specific type of return with care.

You can call a subroutine any number of times in a program. You also can call a subroutine from within another subroutine. How deeply you can nest subroutines is limited only by the available stack space (you can increase the stack space with the **CLEAR** statement). Subroutines that call themselves (recursive subroutines) can easily run out of stack space. **RETURN** with a line label or line number can return control to a statement in the module-level code only, not in procedure-level code. See the example program.

A subroutine can contain more than one **RETURN** statement. A simple **RETURN** statement (without the *linelabel2*, *linenumber2* option) in a subroutine makes BASIC branch back to the statement following the most recent **GOSUB** statement.

Subroutines can appear anywhere in the program, but it is good programming practice to make them readily distinguishable from the main program. To prevent inadvertent entry into a subroutine, precede it with a **STOP**, **END**, or **GOTO** statement that directs program control around the subroutine.

Note The preceding discussion of subroutines applies only to the targets of **GOSUB** statements, not procedures delimited by **SUB** statements. For information on entering and exiting **SUB** procedures, see the entry for the **CALL** statement (BASIC procedures).

See Also

RETURN, **SUB**

Example

The following example shows the use of the **GOSUB** and **RETURN** statements.

Note the **END** statement before the module-level subroutine. If it is not present, the `PrintMessage` subroutine will be executed after the return to `Label1.` BASIC will generate the error message `RETURN without GOSUB.`

```
CLS
GOSUB PrintMessage
PRINT "This message is in module-level code"
GOSUB Sub1
PRINT "This line in module-level code should be skipped."
Label1:
   PRINT "This message is back in module-level code"
END

PrintMessage:
   PRINT "This program uses the GOSUB and RETURN statements."
   PRINT
   RETURN

Sub1:
   PRINT "This message is in Sub1."
   GOSUB Sub2
   PRINT "This line in Sub1 should be skipped."
Label2:
   PRINT "This message is back in Sub1."
   RETURN Label1

Sub2:
   PRINT "This message is in Sub2."
   RETURN Label2    ' Cannot return from here to module-level
                    ' code-only to SUB1.
```

GOTO Statement

Action Branches unconditionally to the specified line.

Syntax **GOTO** {*linelabel* | *linenumber*}

Remarks The argument *linelabel* or *linenumber* indicates the line to execute next. This line must be at the same level of the program. The **GOTO** statement provides a means for branching unconditionally to another line (*linelabel* or *linenumber*). A **GOTO** statement can branch only to another statement at the same level within a program. You cannot use **GOTO** to enter or exit a **SUB** or **FUNCTION** procedure or multiline **DEF FN** function. You can, however, use **GOTO** to control program flow within any of these program structures.

It is good programming practice to use structured control statements (**DO...LOOP**, **FOR...NEXT**, **IF...THEN...ELSE**, **SELECT CASE**) instead of **GOTO** statements, because a program with many **GOTO** statements can be difficult to read and debug.

Example The following example calculates the area of a circle. The program uses the **GOTO** statement to repeat the operation.

```
CLS      ' Clear screen.
PRINT "Input 0 to end."
Start:
   INPUT R
   IF R = 0 THEN
     END
   ELSE
      A = 3.14 * R ^ 2
      PRINT "Area ="; A
   END IF
GOTO Start
```

Output

```
Input 0 to end.
? 5
Area = 78.5
? 0
```

HEX$ Function

Action Returns a string that represents the hexadecimal value of the decimal argument.

Syntax **HEX$**(*numeric-expression&*)

Remarks The argument *numeric-expression&*, which has a decimal value, is rounded to an integer or, if it is outside the integer range, a long integer, before the **HEX$** function evaluates it.

See Also **OCT$**

Example The following example displays the hexadecimal value of a decimal number:

```
CLS          ' Clear screen.
INPUT X
A$ = HEX$(X)
PRINT X; "decimal is "; A$; " hexadecimal."
```

Output

```
? 32
 32 decimal is 20 hexadecimal.
```

IF...THEN...ELSE Statement

Action Allows conditional execution, based on the evaluation of a Boolean expression.

Syntax 1 **IF** *condition* **THEN** *thenpart* [[**ELSE** *elsepart*]]

Syntax 2

IF *condition1* **THEN**
    [[*statementblock-1*]]
 [[**ELSEIF** *condition2* **THEN**
    [[*statementblock-2*]]]]
 [[**ELSE**
    [[*statementblock-n*]]]]
END IF

Remarks

Single-Line IF...THEN...ELSE

The single-line form of the statement (Syntax 1) is best used for short, straightforward tests where only one action is taken.

The single-line form is never required. Any program using single-line **IF...THEN...ELSE** statements can be written using block form.

The following list describes the parts of the single-line form:

Part	Description
condition	Any expression that BASIC evaluates as true (nonzero) or false (zero).
thenpart, *elsepart*	The statements or branches performed when *condition* is true (*thenpart*) or false (*elsepart*). Both parts have the same syntax, which is described below.

The *thenpart* and the *elsepart* fields both have the following syntax:

{*statements* | [[**GOTO**]] *linenumber* | **GOTO** *linelabel* }

The following list describes the parts of the *thenpart* and *elsepart* syntax:

Part	Description
statements	One or more BASIC statements, separated by colons.
linenumber	A valid BASIC program line number.
linelabel	A valid BASIC program line label.

Note that **GOTO** is optional with a line number, but is required with a line label.

The *thenpart* is executed if *condition* is true; if *condition* is false, *elsepart* is executed. If the **ELSE** clause is not present, control passes to the next statement in the program.

You can have multiple statements with a condition, but they must be on the same line and separated by colons, as in the following statement:

```
IF A > 10 THEN A=A+1:B=B+A:LOCATE 10,22:PRINT B,A
```

Block IF...THEN...ELSE

The block form (Syntax 2) provides several advantages:

- The block form provides more structure and flexibility than the single-line form by allowing conditional branches across several lines.
- With the block form, more complex conditions can be tested.
- The block form lets you use longer statements and structures within the **THEN...ELSE** portion of the statement.
- The block form allows your program's structure to be guided by logic rather than by how many statements fit on a line.

Programs that use block **IF...THEN...ELSE** are usually easier to read, maintain, and debug.

The following list describes the parts of the block **IF...THEN...ELSE**:

Part	Description
condition1, *condition2*	Any expression that BASIC evaluates as true (nonzero) or false (zero).
statementblock-1, *statementblock-2*, *statementblock-n*	One or more BASIC statements on one or more lines.

In executing a block **IF**, BASIC tests *condition1*, the first Boolean expression. If the Boolean expression is true (nonzero), the statements following **THEN** are executed. If the first Boolean expression is false (zero), BASIC begins evaluating each **ELSEIF** condition in turn. When BASIC finds a true condition, the statements following the associated **THEN** are executed. If none of the **ELSEIF** conditions is true, the statements following the **ELSE** are executed. After the statements following a **THEN** or **ELSE** are executed, the program continues with the statement following the **END IF**.

The **ELSE** and **ELSEIF** blocks are both optional. You can have as many **ELSEIF** clauses as you would like in a block **IF**. Any of the statement blocks can contain nested block **IF** statements.

BASIC looks at what appears after the **THEN** keyword to determine whether or not an **IF** statement is a block **IF**. If anything other than a comment appears after **THEN**, the statement is treated as a single-line **IF** statement.

A block **IF** statement must be the first statement on a line. The **ELSE**, **ELSEIF**, and **END IF** parts of the statement can have only a line number or line label in front of them. The block *must* end with an **END IF** statement.

For more information, see Chapter 1, "Control-Flow Structures" in the *Programmer's Guide*.

See Also SELECT CASE

Examples The following examples show the use of single-line and block **IF...THEN...ELSE** statements.

Here is the single-line form:

```
CLS    ' Clear screen.
DO
   INPUT "Enter a number greater than 0 and less than 10,000:", X
   IF X >= 0 AND X < 10000 THEN EXIT DO ELSE PRINT X; "out of range"
LOOP
IF X<10 THEN Y=1 ELSE IF X<100 THEN Y=2 ELSE IF X<1000 THEN Y=3 ELSE Y=4
PRINT "The number has"; Y; "digits"
```

Here is the block form, which is easier to read and more powerful:

```
CLS    ' Clear screen.
DO
   INPUT "Enter a number greater than 0 and less than 100,000:", X
   IF X >= 0 AND X < 100000 THEN
      EXIT DO
   ELSE
      PRINT X; "out of range"
   END IF
LOOP

IF X < 10 THEN
   Y = 1
ELSEIF X < 100 THEN
   Y = 2
ELSEIF X < 1000 THEN
   Y = 3
ELSEIF X < 10000 THEN
   Y = 4
ELSE
   Y = 5
END IF
PRINT "The number has"; Y; "digits"
```

$INCLUDE Metacommand

Action Instructs the compiler to include statements from another file.

Syntax 1 **REM $INCLUDE:** '*filespec*'

Syntax 2 **' $INCLUDE:** '*filespec*'

Remarks The argument *filespec* is the name of a BASIC program file, which can include a path. Use single quotation marks around *filespec*. The metacommand **$INCLUDE** instructs the compiler to:

1. Temporarily switch from processing one file.
2. Read program statements from the BASIC file named in *filespec*.
3. Return to processing the original file when the end of the included file is reached.

Because compilation begins with the line immediately following the line in which **$INCLUDE** occurred, **$INCLUDE** should be the last statement on the line. For example:

```
DEFINT I-N  ' $INCLUDE: 'COMMON.BAS'
```

When you are running a program from the QBX environment, included files must not contain **SUB** or **FUNCTION** statements. When you are compiling a program from the BC command line, included files can contain **SUB** or **FUNCTION** statements. Included files created with BASICA must be saved with the ,A option. Included files created with QBX must be in a text (not binary) format.

Example See the **DateSerial#** function programming example, which uses the **$INCLUDE** metacommand. The **DateSerial#** entry is in Part 2, "Add-On-Library Reference."

INKEY$ Function

Action Returns a character from the keyboard or some other standard input device.

Syntax **INKEY$**

Remarks The **INKEY$** function returns a 1- or 2-byte string that contains a character read from the standard input device, which usually is the keyboard. A null string is returned if there is no character to return. A one-character string contains the actual character read from the standard input device, while a two-character string indicates an extended code, the first character of which is hexadecimal 00. For a complete list of these codes, see Appendix A, "Keyboard Scan Codes and ASCII Character Codes."

The character returned by the **INKEY$** function is never displayed on the screen; instead, all characters are passed through to the program except for these key combinations:

Character	Running under QBX	Stand-alone .EXE file
Ctrl+Break	Halts program execution.	Halts program execution if compiled with the /D option; otherwise, passed through to program.
Ctrl+NumLock	Causes program execution to pause until a key is pressed; then the next key pressed is passed through to the program.	Same as running under QBX if compiled with the /D option; otherwise, it is ignored (the program does not pause and no keystroke is passed).
Shift+PrtSc	Prints the screen contents.	Prints the screen contents.
Ctrl+Alt+Del	Reboots the system.	Reboots the system.

If you have assigned a string to a function key using the **KEY** statement and you press that function key when **INKEY$** is waiting for a keystroke, **INKEY$** passes the string to the program. Enabled keystroke trapping takes precedence over the **INKEY$** function.

When you use **INKEY$** with ISAM programs, BASIC performs implicit **CHECKPOINT** operations to minimize data loss in the event of a power failure. The **CHECKPOINT** is performed if **INKEY$** fails to successfully retrieve a character after 65,535 calls, and 20 seconds has expired. A **CHECKPOINT** writes open database buffers to disk.

Example The following example shows a common use of **INKEY$**. The program pauses until the user presses a key:

```
PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$=""
```

INP Function

Action Returns the byte read from a hardware I/O port.

Syntax INP(*port*)

Remarks The **INP** function complements the **OUT** statement, which sends a byte to a hardware I/O port.

The argument *port* identifies the hardware I/O port from which to read the byte. It must be a numeric expression with an integer value between 0 and 65,535, inclusive.

The **INP** function and the **OUT** statement give a BASIC program direct control over the system hardware through the I/O ports. **INP** and **OUT** must be used carefully because they directly manipulate the system hardware.

Note The **INP** function is not available in OS/2 protected mode.

See Also **OUT, WAIT**

Example See the **OUT** statement programming example, which uses the **INP** function.

INPUT$ Function

Action Returns a string of characters read from the specified file.

Syntax **INPUT$(***n* ⟦, ⟦#⟧*filenumber%* ⟧)

Remarks The **INPUT$** statement uses the following arguments:

Argument	Description
n	The number of characters (bytes) to read from the file.
filenumber%	The number that was used in the **OPEN** statement to open the file.

If the file is opened for random access, the argument *n* must be less than or equal to the record length set by the **LEN** clause in the **OPEN** statement (or less than or equal to 128 if the record length is not set). If the given file is opened for binary or sequential access, *n* must be less than or equal to 32,767.

If *filenumber%* is omitted, the characters are read from the standard input device. (If input has not been redirected, the keyboard is the standard input device.)

You can use the DOS redirection symbols (<, >, or >>) or the pipe symbol (|) to redefine the standard input or standard output for an executable file created with BASIC. (See your operating-system manual for a complete discussion of redirection and pipes.)

Unlike the **INPUT #** statement, **INPUT$** returns all characters it reads.

Example The following example uses **INPUT$** to read one character at a time from a file. The input character is converted to standard printable ASCII, if necessary, and displayed on the screen.

```
' ASCII codes for tab and line feed.
DEFINT A-Z
CONST HTAB = 9, LFEED = 10
CLS    ' Clear screen.
INPUT "Display which file"; Filename$
OPEN Filename$ FOR INPUT AS #1
DO WHILE NOT EOF(1)
    ' Input a single character from the file.
    S$ = INPUT$(1, #1)
    ' Convert the character to an integer and turn off the high bit
    ' so WordStar files can be displayed.
    C = ASC(S$) AND &H7F
    ' Is it a printable character?
    IF (C >= 32 AND C <= 126) OR C = HTAB OR C = LFEED THEN PRINT CHR$(C);
LOOP
END
```

INPUT Statement

Action Allows input from the keyboard during program execution.

Syntax **INPUT** ⟦;⟧ ⟦"*promptstring*"{;|,}⟧ *variablelist*

Remarks The following list describes the parts of the **INPUT** statement:

Part	Description
;	Determines screen location of the text cursor after the program user enters the last character of input. A semicolon immediately after **INPUT** keeps the cursor at the next character position. Otherwise, the cursor skips to the next line.
promptstring	A literal string that will be displayed on the screen before the program user enters data items into the data input field.
;	Prints a question mark at the end of *promptstring*.
,	Prints *promptstring* without a question mark.
variablelist	An ordered list of variables that will hold the data items as they are entered.

The **INPUT** statement causes a running program to pause and wait for the user to enter data from the keyboard.

The number and type of data items required from the user is determined by the structure of *variablelist*. The variables in *variablelist* can be made up of one or more variable names, each separated from the next by a comma. Each name may refer to:

- A simple numeric variable.
- A simple string variable.
- A numeric or string array element.
- A record element.

The number of entered data items must be the same as the number of variables in the list. The type of each entered data item must agree with the type of the variable. If either of these rules is violated, the program will display the prompt `Redo from start` and no assignment of input values will be made.

The **INPUT** statement determines the number of entered data items in a list as follows: The first character encountered after a comma that is not a space, carriage return, or line feed is assumed to be the start of a new item. If this first character is a quotation mark ("), the item is typed as string data and will consist of all characters between the first quotation mark and the second. Not all strings have to start with a quotation mark, however. If the first character of the string is not a quotation mark, it terminates on a comma, carriage return, or line feed.

Input stored in a record must be entered as single elements. For example:

```
TYPE Demograph
   FullName AS STRING * 25
   Age AS INTEGER
END TYPE
DIM Person AS Demograph
INPUT "Enter name and age: "; Person.FullName, Person.Age
```

You may want to instruct the user on how it is possible to edit a line of input before pressing the Enter key to submit the line to the program. The following list describes the key combinations that allow you to move the cursor, delete text, and insert text on the input line:

Keystrokes	Edit function
Ctrl+M or Enter	Store input line.
Ctrl+H or Backspace	Delete the character to the left of the cursor, unless the cursor is at the beginning of the input, in which case it deletes the character at the cursor.
Ctrl+\ or Right Arrow	Move cursor one character to the right.
Ctrl+] or Left Arrow	Move cursor one character to the left.
Ctrl+F or Ctrl+Right Arrow	Move cursor one word to the right.
Ctrl+B or Ctrl+Left Arrow	Move cursor one word to the left.
Ctrl+K or Home	Move cursor to beginning of input line.
Ctrl+N or End	Move cursor to end of input line.
Ctrl+R or Ins	Toggle insert mode on and off.
Ctrl+I or Tab	Tab right and insert (insert mode on), or tab right and overwrite (insert mode off).
Del	Delete the character at the cursor.
Ctrl+E or Ctrl+End	Delete to the end of the line.
Ctrl+U or Esc	Delete entire line, regardless of cursor position.
Ctrl+T	Toggle function key label display on and off at bottom of screen.
Ctrl+C or Ctrl+Break	Terminate input (exit compiled program).

When you use **INPUT** with ISAM programs, BASIC performs a **CHECKPOINT** operation every 20 seconds during an **INPUT** polling loop. A **CHECKPOINT** writes open database buffers to disk.

If the user can be limited to using the Enter and Backspace keys for editing, you can reduce the size of the .EXE file by linking with the stub file NOEDIT.OBJ.

If the user will be entering decimal integers only, you can save between 1.6K and 11K in the size of a non-stand-alone .EXE file by linking with the stub file NOFLTIN.OBJ. This places the following restrictions on user input:

- Decimal numbers only (no leading &H, &O, or & base specifiers).
- No trailing type specifiers (%, &, !, #, @, or $).
- No decimal point, E, or D (for example 1.E2 cannot be used instead of the integer 100).

See Also

INKEY$

Example

This example calculates the area of a circle from its radius. The program uses the **INPUT** statement to allow the user to supply values of the variable.

```
CLS
PI = 3.141593: R = -1
DO WHILE R
   PRINT "Enter radius (or 0 to quit)."
   INPUT ; "If radius = ", R
   IF R > 0 THEN
      A = PI * R ^ 2
      PRINT ", the area of the circle ="; A
   END IF
   PRINT
LOOP
```

Output

```
Enter radius (or 0 to quit).
If radius = 3, the area of the circle = 28.27434

Enter radius (or 0 to quit).
If radius = 4, the area of the circle = 50.26549

Enter radius (or 0 to quit).
If radius = 0
```

INPUT # Statement

Action Reads data items from a sequential device or file and assigns them to variables.

Syntax **INPUT** #*filenumber%*, *variablelist*

Remarks The argument *filenumber%* is the number used in the **OPEN** statement to open the file. The argument *variablelist* contains the names of the variables that are assigned values read from the file.

The data items in the file should appear just as they would if you were entering data in response to an **INPUT** statement. Separate numbers with a space, carriage return, line feed, or comma. Separate strings with a carriage return or line feed (leading spaces are ignored). The end-of-file character will end either a numeric or string entry.

See Also **INPUT$** Function, **INPUT** Statement

Example This example reads a series of test scores from a sequential file. It then calculates and displays the average score.

```
DEFINT A-Z
DIM Total AS LONG
' Create the required input file.
OPEN "class.dat" FOR OUTPUT AS #1
PRINT #1, 98, 84, 63, 89, 100
CLOSE #1

' Open the file just created.
OPEN "class.dat" FOR INPUT AS #1
DO WHILE NOT EOF(1)     'Do until end-of-file is reached.
    Count = Count + 1
    INPUT #1, Score     'Get a score from file #1.
    Total = Total + Score
    PRINT Count; Score
LOOP
PRINT "Total students:"; Count; " Average score:"; Total / Count
```

Output

```
1  98
2  84
3  63
4  89
5  100
Total students: 5  Average score: 86.8
```

INSERT Statement

Action Adds a new record to an ISAM table.

Syntax **INSERT** [[#]]*filenumber%*,*recordvariable*

Remarks The **INSERT** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the table.
recordvariable	The record you wish to insert. It is a variable of the user-defined type corresponding to the table.

INSERT places the contents of *recordvariable* in the table, and updates the table's indexes to include the new record. **INSERT** has no effect on the current position.

BASIC removes trailing spaces from strings used in an insert.

See Also **DELETE**, **OPEN** (File I/O), **RETRIEVE**

Example See the programming example for the **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements, which uses the **INSERT** statement.

INSTR Function

Action Returns the character position of the first occurrence of a string in another string.

Syntax **INSTR**([[*start%*,]]*stringexpression1$*, *stringexpression2$*)

Remarks The **INSTR** function uses the following arguments:

Argument	Description
start%	An optional offset that sets the position for starting the search; *start%* must be between 1 and 32,767, inclusive. If *start%* is not given, the **INSTR** function begins the search at the first character of *stringexpression1$*.
stringexpression1$	The string being searched.
stringexpression2$	The string to look for.

The arguments *stringexpression1$* and *stringexpression2$* can be string variables, string expressions, or string literals. The value returned by **INSTR** depends on these conditions:

Condition	Value returned
stringexpression2$* found in *stringexpression1$	The position at which the match is found
start% greater than length of *stringexpression1$*	0
stringexpression1$ is a null string	0
stringexpression2$ cannot be found	0
stringexpression2$ is a null string	*start%* (if given); otherwise, 1

Use the **LEN** function to find the length of *stringexpression1$*.

See Also **LEN**

Example The following example uses **INSTR** and **UCASE$** to determine a person's gender from a courtesy title (Mr., Mrs., or Ms.):

```
CLS     ' Clear screen.
DO
   INPUT "Enter name with courtesy title (Mr., Mrs., or Ms.): ", Nm$
LOOP UNTIL LEN(Nm$) >= 3

Nm$ = UCASE$(Nm$)       ' Convert lowercase letters to uppercase.
' Look for MS, MRS, or MR to set Sex$.
IF INSTR(Nm$, "MS") > 0 OR INSTR(Nm$, "MRS") > 0 THEN
    Sex$ = "F"
ELSEIF INSTR(Nm$, "MR") > 0 THEN
    Sex$ = "M"
ELSE
    ' Can't determine gender, query user.
    DO
        INPUT "Enter sex (M/F): ", Sex$
        Sex$ = UCASE$(Sex$)
    LOOP WHILE Sex$ <> "M" AND Sex$ <> "F"
END IF
PRINT "Sex is "; Sex$
```

Output

```
Enter name: Ms. Elspeth Brandtkeep
Sex is F

Enter name: Dr. Richard Science
Enter sex (M/F): M
Sex is M
```

INT Function

Action Returns the largest integer less than or equal to a numeric expression.

Syntax **INT**(*numeric-expression*)

Remarks The **INT** function removes the fractional part of its argument.

See Also **CINT, FIX**

Example The following example compares output from **INT**, **CINT**, and **FIX**, the three functions that convert numeric data to integers:

```
CLS                  ' Clear screen.
PRINT "  N","INT(N)","CINT(N)","FIX(N)" : PRINT
FOR I% = 1 TO 6
    READ N
    PRINT N, INT(N), CINT(N), FIX(N)
NEXT
DATA  99.3, 99.5, 99.7, -99.3, -99.5, -99.7
```

Output

```
 N            INT(N)        CINT(N)       FIX(N)

 99.3          99            99            99
 99.5          99            100           99
 99.7          99            100           99
-99.3         -100          -99           -99
-99.5         -100          -100          -99
-99.7         -100          -100          -99
```

Interrupt, InterruptX Routines

Action Allow BASIC programs to perform DOS system calls.

Syntax CALL **Interrupt** (*interruptnum%*, *inregs*, *outregs*)
CALL **InterruptX** (*interruptnum%*, *inregs*, *outregs*)

Remarks The **Interrupt** routines use the following arguments:

Argument	Description
interruptnum%	A DOS interrupt number that is an integer between 0 and 255.
inregs	The register values before the interrupt is performed; *inregs* is declared as type `RegType` or `RegTypeX`. The user-defined types `RegType` and `RegTypeX` are described below.
outregs	The register values after the interrupt is performed; *outregs* is declared as type `RegType` or `RegTypeX`. The user-defined types `RegType` and `RegTypeX` are described below.

The following statement defines `RegTypeX`:

```
TYPE RegTypeX
   AX AS INTEGER
   BX AS INTEGER
   CX AS INTEGER
   DX AS INTEGER
   BP AS INTEGER
   SI AS INTEGER
   DI AS INTEGER
   FLAGS AS INTEGER
   DS AS INTEGER
   ES AS INTEGER
END TYPE
```

The following statement defines `RegType` (the DS and ES registers are not included):

```
TYPE RegType
   AX AS INTEGER
   BX AS INTEGER
   CX AS INTEGER
   DX AS INTEGER
   BP AS INTEGER
   SI AS INTEGER
   DI AS INTEGER
   FLAGS AS INTEGER
END TYPE
```

Each element of the type corresponds to a CPU element.

InterruptX uses the values in the DS and ES registers. To use the current values of these registers, set the record elements to –1.

The Interrupt and **InterruptX** routines replace the INT86 and INT86X routines used in earlier versions of BASIC. They provide a more convenient way for BASIC programs to use DOS interrupts and services.

To use **Interrupt** or **InterruptX** in the QBX environment, use the QBX.QLB Quick library. To use **Interrupt** or **InterruptX** outside of the QBX environment, link your program with the QBX.LIB file.

The QBX.BI header file contains the necessary declarations for **Interrupt** and **InterruptX** .

Note

The **Interrupt** and **InterruptX** routines are not available in OS/2 protected mode. For OS/2 protected mode, replace **Interrupt** with the equivalent OS/2 function invocations. For more information about doing OS/2 calls from BASIC, see Chapter 14, "OS/2 Programming" in the *Programmer's Guide*.

Example

The following example uses **Interrupt** to determine the current drive and the amount of free space remaining on the drive.

To use **Interrupt**, you must load the Quick library QBX.QLB using the /L option when you begin QBX. You also must include the QBX.BI header file as shown below:

```
' $INCLUDE: 'QBX.BI'
DIM regs AS RegType    ' Define registers.

' Get current drive info.
regs.ax = &H1900
CALL Interrupt(&H21, regs, regs)
' Convert drive info to readable form.
Drive$ = CHR$((regs.ax AND &HFF) + 65) + ":"
```

```
' Get disk's free space.
regs.ax = &H3600
regs.dx = ASC(UCASE$(Drive$)) - 64
CALL Interrupt(&H21, regs, regs)
' Decipher the results.
SectorsInCluster = regs.ax
BytesInSector = regs.cx
IF regs.dx >= 0 THEN
   ClustersInDrive = regs.dx
ELSE
   ClustersInDrive = regs.dx + 65536
END IF
IF regs.bx >= 0 THEN
   ClustersAvailable = regs.bx
ELSE
   ClustersAvailable = regx.bx + 65536
END IF
Freespace = ClustersAvailable * SectorsInCluster * BytesInSector

' Report results.
CLS
PRINT "Drive "; Drive$; " has a total of";
PRINT USING "###,###,###"; Freespace;
PRINT " bytes remaining free."
```

IOCTL$ Function

Action Returns current status information from a device driver.

Syntax **IOCTL$**([[#]]*filenumber%*)

Remarks The argument *filenumber%* is the BASIC file number used to open the device.

The **IOCTL$** function is most frequently used to test whether an **IOCTL** statement succeeded or failed, or to obtain current status information.

You can use **IOCTL$** to ask a communications device to return the current baud rate, information on the last error, logical line width, and so on. The exact information returned depends on the specific device driver. See the device driver documentation to find out what status information the device driver can send.

The **IOCTL$** function works only if all of the following conditions are met:

- The device driver is installed.
- The device driver states that it processes **IOCTL** strings. See the documentation for the driver.
- BASIC performed an **OPEN** operation on a file on that device, and the file is still open.

Most standard DOS device drivers do not process **IOCTL** strings, and you must determine whether the specific driver accepts the command. If the driver does not process **IOCTL** strings, BASIC generates the error message `Illegal function call.`

Note BASIC devices (LPT*n*, COM*n*, SCRN, CONS, PIPE) and DOS block devices (A through Z) do not support **IOCTL**.

The **IOCTL** statement is not available in OS/2 protected mode. However, you can achieve the same effect by directly invoking the DosDevIOCtl OS/2 functions.

See Also **IOCTL** Statement

Example The following example shows how to communicate with a device driver using a hypothetical device driver named `ENGINE`. The **IOCTL** statement sets the data mode in the driver and the **IOCTL$** function tests the data mode.

```
OPEN "\DEV\ENGINE" FOR OUTPUT AS #1
IOCTL #1, "RAW"     ' Tells the device that the data is raw.

' If the character driver "ENGINE" responds "false" from the raw data
' mode in the IOCTL statement, then the file is closed.
IF IOCTL$(1) = "0" THEN CLOSE 1
```

IOCTL Statement

Action Transmits a control data string to a device driver.

Syntax **IOCTL** [[#]]*filenumber%*,*string$*

Remarks The **IOCTL** statement uses the following arguments:

Argument	Description
filenumber%	The BASIC file number used to open the device.
string$	The command sent to the device.

Commands are specific to the device driver. See the documentation for the device driver for a description of the valid **IOCTL** commands. An **IOCTL** control data string can be up to 32,767 bytes long.

The **IOCTL** statement works only if all of the following conditions are met:

- The device driver is installed.
- The device driver states that it processes **IOCTL** strings. See the documentation for the driver. You also can test for **IOCTL** support through DOS function &H44 by using interrupt &H21 and the **INTERRUPT** routine. For more information about interrupt &H21, function &H44, see see the *Microsoft MS-DOS Programmer's Reference*, or books such as *Advanced MS-DOS* or *The Peter Norton Guide to the IBM PC*.
- BASIC performed an **OPEN** operation on a file on that device, and the file is still open.

Most standard DOS device drivers do not process **IOCTL** strings, and you must determine whether the specific driver accepts the command. If the driver does not process **IOCTL** strings, BASIC generates the error message `Illegal function call.`

Note BASIC devices (LPT*n*, COM*n*, SCRN, CONS, PIPE) and DOS block devices (A through Z) do not support **IOCTL**.

The **IOCTL** statement is not available in OS/2 protected mode. However, you can achieve the same effect by directly invoking the DosDevIOCtl OS/2 function.

See Also IOCTL$ Function

Example See the **IOCTL$** function programming example, which shows how the **IOCTL** statement and the **IOCTL$** function are used with a device driver.

KEY Statements (Assignment)

Action

Assign soft-key string values to function keys, then display the values and enable or disable the function-key display line.

Syntax

KEY *n%*, *stringexpression$*
KEY ON
KEY OFF
KEY LIST

Remarks

The argument *n%* is a number representing the function key. The values for *n%* are 1 to 10 for the function keys, and 30 and 31 for function keys F11 and F12 on 101-key keyboards. The *stringexpression$* is a string of up to 15 characters that is returned when the function key is pressed. If the *stringexpression$* is longer than 15 characters, the extra characters are ignored.

The **KEY** statements allows you to designate special "soft-key" functions—strings that are returned when function keys are pressed.

Assigning a null string to a soft key disables the function key as a soft key.

If the function key number is not in the correct range, BASIC generates the error message `Illegal function call`, and the previous key string expression is retained.

You can display soft keys with the **KEY ON**, **KEY OFF**, and **KEY LIST** statements:

Statement	Action
KEY ON	Displays the first six characters of the soft-key string values on the bottom line of the screen.
KEY OFF	Erases the soft-key display from the bottom line, making that line available for program use. It does not disable the function keys.
KEY LIST	Displays all soft-key values on the screen, with all 15 characters of each key displayed.

If a soft key is pressed, the effect is the same as if the user typed the string associated with the soft key. **INPUT$, INPUT**, and **INKEY$** all can be used to read the string produced by pressing the soft key.

See Also

ON *event*; Appendix A, "Keyboard Scan Codes and ASCII Character Codes"

Examples

The following examples show how to assign values to soft keys. First is an example of assigning and disabling a soft key. **KEY LIST** displays key values after `KEY 4` has been assigned and again after it has been disabled.

```
KEY 4, "MENU" + CHR$(13)      ' Assigns to soft key 4 the string
KEY LIST                      ' "MENU" followed by a carriage return.
KEY 4, ""                     ' Disables soft key 4.
KEY LIST
```

This is an example of using **KEY** statements to set up one-key equivalents of menu selections. For example, pressing the F1 key is the same as entering the string "Add":

```
CLS                          ' Clear screen.
DIM KeyText$(3)
DATA Add, Delete, Quit
' Assign soft-key strings to F1 to F3.
FOR I = 1 TO 3
   READ KeyText$(I)
   KEY I, KeyText$(I) + CHR$(13)    ' String followed by Enter.
NEXT I

' Print menu.
PRINT "                 Main Menu" : PRINT
PRINT "           Add to list (F1)"
PRINT "           Delete from list (F2)"
PRINT "           Quit (F3)" : PRINT

' Get input and respond.
DO
   LOCATE 7,1 : PRINT SPACE$(50);
   LOCATE 7,1 : INPUT "            Enter your choice:", R$
   SELECT CASE R$
      CASE "Add", "Delete"
         LOCATE 10,1 : PRINT SPACE$(15);
         LOCATE 10,1 : PRINT R$;
      CASE "Quit"
         EXIT DO
      CASE ELSE
         LOCATE 10,1 : PRINT "Enter first word or press key."
   END SELECT
LOOP
```

KEY Statements (Event Trapping)

Action Enable, disable, or suspend trapping of specified keys.

Syntax

KEY(*n%*) **ON**
KEY(*n%*) **OFF**
KEY(*n%*) **STOP**

Remarks The argument *n%* is an integer expression that is the number of a function key, a direction key, or a user-defined key. The values of *n%* are as follows:

n%	**Value**
0	All keys listed in this table
1–10	F1–F10
11	Up Arrow key
12	Left Arrow key
13	Right Arrow key
14	Down Arrow key
15–25	User-defined keys
30–31	F11–F12 on 101-key keyboards

The **KEY**(*n%*) **ON** statement enables trapping of function keys, direction keys, and user-defined keys. If key *n%* is pressed after a **KEY**(*n%*) **ON** statement, the routine specified in the **ON KEY** statement is executed.

KEY(*n%* **OFF** disables trapping of key *n%*. No key trapping takes place until another **KEY**(*n%*) **ON** statement is executed. Events occurring while trapping is off are ignored.

KEY(*n%*) **STOP** suspends trapping of key *n%*. No trapping takes place until a **KEY**(*n%*) **ON** statement is executed. Events occurring while trapping is suspended are remembered and processed when the next **KEY**(*n%*) **ON** statement is executed. However, remembered events are lost if **KEY**(*n%*) **OFF** is executed.

When a key-event trap occurs (that is, the **GOSUB** is performed), an automatic **KEY**(*n%*) **STOP** is executed so that recursive traps cannot take place. The **RETURN** operation from the trapping routine automatically performs a **KEY**(*n%*) **ON** statement unless an explicit **KEY**(*n%*) **OFF** was performed inside the subroutine.

For more information on event trapping, see Chapter 9, "Event Handling" in the *Programmer's Guide*.

In addition to providing the preassigned key numbers 1-14 (plus 30 and 31 on the 101-key keyboard), BASIC enables you to create user-defined keys. You do this by assigning the numbers 15-25 to any of the remaining keys on the keyboard. Use the **KEY** statement (assignment) to create user-defined keys.

You also can set a trap for "shifted" keys. A key is shifted when you press it simultaneously with one or more of the special keys Shift, Ctrl, or Alt after pressing NumLock or Caps Lock. Use the **KEY** statement (assignment) to define shifted keys before you trap them. The syntax for **KEY** (assignment) is:

KEY *n%*, **CHR$**(*keyboardflag*) + **CHR$**(*scancode*)

The argument *n%* is in the range 15-25 to indicate a user-defined key. The argument *keyboardflag* can be any combination of the following values:

Value	Key
0	No keyboard flag
1, 2, or 3	Either Shift key
4	Ctrl
8	Alt
32	NumLock
64	Caps Lock
128	101-key keyboard extended keys

You can add the values together to test for multiple shift states. A *keyboardflag* value of 12 would test for both Ctrl and Alt being pressed, for example.

To define Shift, Ctrl, Alt, NumLock, or Caps Lock as a user-defined key (by itself, not in combination with another key), use a keyboard flag of 0. For example, to define Alt as a user-defined key, use the following statement:

```
KEY CHR$(0) + CHR$(56)
```

To define Alt + Alt as a user-defined key (the second Alt will be trapped when it is pressed only if the first Alt key is already being pressed), use the following statement:

```
KEY CHR$(8) + CHR$(56)
```

Because key trapping assumes the left and right Shift keys are the same, you can use 1, 2, or 3 to indicate a Shift key.

The argument *scancode* is a number that identifies one of the 83 keys to trap, as shown in the following table:

Keyboard Scan Codes

Key	Code	Key	Code	Key	Code
Esc	1	Ctrl	29	Spacebar	57
! or 1	2	A	30	Caps Lock	58
@ or 2	3	S	31	F1	59
# or 3	4	D	32	F2	60
$ or 4	5	F	33	F3	61
% or 5	6	G	34	F4	62
^ or 6	7	H	35	F5	63
& or 7	8	J	36	F6	64
* or 8	9	K	37	F7	65
(or 9	10	L	38	F8	66
) or 0	11	: or ;	39	F9	67
_ or –	12	" or '	40	F10	68
+ or =	13	~ or `	41	NumLock	69
Backspace	14	Left Shift	42	Scroll Lock	70
Tab	15		or \	43	Home or 7	71
Q	16	Z	44	Up or 8	72
W	17	X	45	PgUp or 9	73
E	18	C	46	Gray -	74
R	19	V	47	Left or 4	75
T	20	B	48	Center or 5	76
Y	21	N	49	Right or 6	77
U	22	M	50	Gray +	78
I	23	< or ,	51	End or 1	79
O	24	> or .	52	Down or 2	80
P	25	? or /	53	PgDn or 3	81
{ or [	26	Right Shift	54	Ins or 0	82
} or]	27	PrtSc or *	55	Del or .	83
Enter	28	Alt	56		

Note The scan codes in the preceding table are equivalent to the first column of the scan code table in Appendix A, "Keyboard Scan Codes and ASCII Character Codes." The codes in the other columns of the table in the appendix should not be used for key trapping.

See Also **KEY** (Assignment), **ON** *event*

Example The following example traps the Down Arrow key and Ctrl+s (Control key and lowercase "s"). To trap the combination of the Ctrl key and uppercase "s", trap Ctrl+Shift and Ctrl+Caps Lock+s.

```
I = 0
CLS          ' Clear screen.
PRINT "Press Down Arrow key to end."
KEY 15, CHR$(&H4) + CHR$(&H1F)
KEY(15) ON          ' Trap Ctrl+s.
KEY(14) ON          ' Trap Down Arrow key.
ON KEY(15) GOSUB Keytrap
ON KEY(14) GOSUB Endprog
Idle: GOTO Idle     ' Endless loop.

Keytrap:            ' Counts the number of times Ctrl+s pressed.
   I = I + 1
RETURN

Endprog:
   PRINT "CTRL+s trapped"; I; "times"
   END
RETURN
```

KILL Statement

Action Deletes files from a disk.

Syntax **KILL** *filespec$*

Remarks The **KILL** statement is similar to the DOS ERASE or DEL command.

KILL is used for all types of disk files: program files, random-access data files, and sequential data files. The *filespec$* is a string expression that can contain a path and question marks (?) or asterisks (*) used as DOS wildcards. A question mark matches any single character in the filename or extension. An asterisk matches one or more characters.

KILL deletes files only. To delete directories, use the DOS RMDIR command or BASIC **RMDIR** statement. Using **KILL** to delete a file that is currently open produces an error message that reads `File already open.`

Warning Be extremely careful when using wildcards with **KILL**. You can delete files unintentionally with the wildcard characters.

See Also **FILES**, **RMDIR**

Examples The first example uses wildcard characters with **KILL**. It will not work properly unless the specified files are found.

```
KILL "DATA1?.DAT"   ' Kills any file with a six-character
                    ' base name starting with DATA1 and
                    ' also with the extension .DAT.

KILL "DATA1.*"      ' Kills any file with the base name
                    ' DATA1 and any extension.

KILL "\GREG\*.DAT"  ' Kills any file with the extension
                    ' .DAT in a subdirectory called GREG.
```

The following example deletes the file specified on the command line:

```
DEFINT A-Z
CLS                              ' Clear screen.
ON ERROR GOTO Errorhandle        ' Set up error handling.
FileName$ = COMMAND$             ' Get filename.
KILL FileName$
PRINT FileName$ " deleted"
END

Errorhandle:
    Number = ERR
    IF Number = 53 THEN
       PRINT "Couldn't delete " FileName$ ;
       PRINT "; file does not exist in current directory"
    ELSE
       PRINT "Unrecoverable error:";Number
       ON ERROR GOTO 0 ' ON ERROR GOTO zero aborts program.
    END IF
RESUME NEXT
```

LBOUND Function

Action Returns the lower bound (smallest available subscript) for the indicated dimension of an array.

Syntax **LBOUND**(*array* ⟦, *dimension%* ⟧)

Remarks The **LBOUND** function is used with the **UBOUND** function to determine the size of an array. **LBOUND** takes the following arguments:

Argument	Description
array	The name of the array variable to be tested.
dimension%	An integer ranging from 1 to the number of dimensions in *array*; indicates which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second dimension, and so on. This argument is optional for one-dimensional arrays.

LBOUND returns the values listed below for an array with the following dimensions:

```
DIM A(1 TO 100, 0 TO 3, -3 TO 4)
```

Invocation	Value returned
`LBOUND(A,1)`	1
`LBOUND(A,2)`	0
`LBOUND(A,3)`	–3

The default lower bound for any dimension is either 0 or 1, depending on the setting of the **OPTION BASE** statement. If **OPTION BASE** is 0, the default lower bound is 0, and if **OPTION BASE** is 1, the default lower bound is 1.

Arrays dimensioned using the **TO** clause in the **DIM** statement can have any integer value as a lower bound.

You can use the shortened syntax **LBOUND**(*array*) for one-dimensional arrays, because the default value for *dimension%* is 1. Use the **UBOUND** function to find the upper limit of an array dimension.

See Also **DIM**, **OPTION BASE**, **UBOUND**

Example See the **UBOUND** function programming example, which uses the **LBOUND** function.

LCASE$ Function

Action Returns a string with all letters in lowercase.

Syntax **LCASE$** (*stringexpression$*)

Remarks The **LCASE$** function takes a string variable, string constant, or string expression as its single argument. **LCASE$** works with both variable- and fixed-length strings.

LCASE$** and **UCASE$ are helpful in making string comparisons that are not case sensitive.

See Also **UCASE$**

Example The following example converts uppercase characters in a string to lowercase:

```
CLS                ' Clear screen.
READ Word$
PRINT LCASE$(Word$);
DATA "THIS IS THE STRING in lower case."
```

Output

```
this is the string in lower case.
```

LEFT$ Function

Action Returns a string consisting of the leftmost *n* characters of a string.

Syntax **LEFT$**(*stringexpression$*, *n%*)

Remarks The argument *stringexpression$* can be any string variable, string constant, or string expression.

The argument *n%* is a numeric expression in the range 0–32,767 indicating how many characters are to be returned.

If *n%* is 0, the null string (length zero) is returned.

If *n%* is greater than or equal to the number of characters in *stringexpression$*, the entire string is returned. To find the number of characters in *stringexpression$*, use **LEN**(*stringexpression$*).

See Also **MID$** Function, **RIGHT$** Function

Example The following example prints the leftmost five characters of `A$`:

```
CLS                     ' Clear screen.
A$="BASIC LANGUAGE"
B$=LEFT$(A$, 5)
PRINT B$
```

Output

```
BASIC
```

LEN Function

Action Returns the number of characters in a string or the number of bytes required by a variable.

Syntax 1 LEN(*stringexpression$*)

Syntax 2 LEN(*variable*)

Remarks In the first form, **LEN** returns the number of characters in the argument *stringexpression$*. The second syntax returns the number of bytes required by a BASIC variable. This syntax is particularly useful for determining the correct record size of a random-access file.

Example The following example prints the length of a string and the size in bytes of several types of variables:

```
CLS    ' Clear screen.
TYPE EmpRec
   EmpName AS STRING * 20
   EmpNum AS INTEGER
END TYPE
DIM A AS INTEGER, B AS LONG, C AS SINGLE, D AS DOUBLE
DIM E AS EmpRec

PRINT "A string:" LEN("A string.")
PRINT "Integer:" LEN(A)
PRINT "Long:" LEN(B)
PRINT "Single:" LEN(C)
PRINT "Double:" LEN(D)
PRINT "EmpRec:" LEN(E)
END
```

Output

```
A string: 9
Integer: 2
Long: 4
Single: 4
Double: 8
EmpRec: 22
```

LET Statement

Action Assigns the value of an expression to a variable.

Syntax [[LET]] *variable=expression*

Remarks Notice that the keyword **LET** is optional. The equal sign in the statement is enough to inform BASIC that the statement is an assignment statement.

The **LET** statement uses the following arguments:

Argument	Description
variable	A variable.
expression	An expression that provides the value to assign to the variable.

LET statements can be used with record variables only when both variables are the same user-defined type. Use the **LSET** statement to assign record variables of different user-defined types.

See Also **LSET**

Examples The first example below shows the use of the optional **LET** keyword:

```
LET D = 12
LET E = 12 - 2
LET F = 12 - 4
LET SUM = D + E + F
PRINT D E F SUM
```

The following program lines perform the same function, without using the **LET** keyword:

```
D = 12
E = 12 - 2
F = 12 - 4
SUM = D + E + F
PRINT D E F SUM
```

Output

```
12 10 8 30
```

LINE Statement

Action Draws a line or box on the screen.

Syntax **LINE** ⟦⟦**STEP**⟧ (*x1!,y1!*)⟧ – ⟦**STEP**⟧ (*x2!,y2!*) ⟦, ⟦*color&*⟧ ⟦, ⟦B ⟦F⟧⟧ ⟦, *style%*⟧⟧⟧

Remarks The coordinates (*x1!,y1!*) and (*x2!,y2!*) specify the end points of the line; note that the order in which these end points appear is unimportant, since a line from (10,20) to (120,130) is the same as a line from (120,130) to (10,20).

The **STEP** option makes the specified coordinates relative to the most-recent point. For example, if the most-recent point referred to by the program were (10,10), then the following statement would draw a line from (10,10) to the point with an x coordinate equal to 10 + 10 and a y coordinate equal to 10 + 5, or (20,15):

```
LINE -STEP (10,5)
```

You may establish a new most-recent point by initializing the screen with the **CLS** and **SCREEN** statements. Using the **PSET**, **PRESET**, **CIRCLE**, and **DRAW** statements will also establish a new most-recent point.

Variations of the **STEP** argument are shown below. For the following examples, assume that the last point plotted was (10,10):

Statement	Description
`LINE -(50,50)`	Draws from (10,10) to (50,50).
`LINE -STEP(50,50)`	Draws from (10,10) to (60,60).
`LINE (25,25)-STEP(50,50)`	Draws from (25,25) to (75,75).
`LINE STEP(25,25)-STEP(50,50)`	Draws from (35,35) to (85,85).
`LINE STEP(25,25)-(50,50)`	Draws from (35,35) to (50,50).

The argument *color%* is the number of the color in which the line is drawn. (If the B or BF option is used, the box is drawn in this color.) For information on valid colors, see the **SCREEN** statement.

The B option draws a box with the points (*x1!,y1!*) and (*x2!,y2!*) specifying diagonally opposite corners.

The BF option draws a filled box. This option is similar to the B option; BF also paints the interior of the box with the selected color.

The argument *style%* is a 16-bit integer mask used to put pixels on the screen. Using *style%* is called "line styling." With line styling, **LINE** reads the bits in *style%* from left to right. If a bit is 0, then no point is plotted; if the bit is 1, a point is plotted. After plotting a point, **LINE** selects the next bit position in *style%*.

Because a zero bit in *style%* does not change the point on the screen, you may want to draw a background line before using styling so you can have a known background. Style is used for normal lines and boxes, but has no effect on filled boxes.

When coordinates specify a point that is not within the current viewport, the line segment to that point is drawn to the border of the viewport.

For more information on the **LINE** statement, see Chapter 5, "Graphics" in the *Programmer's Guide*.

See Also

CIRCLE, **PSET**, **SCREEN** Statement

Examples

The following example uses **LINE** statements to display a series of screens with different line graphics. To run this program, your screen must be 320 pixels wide by 200 pixels high and support CGA screen mode.

```
SCREEN 1                        ' Set up the screen mode.

LINE -(X2, Y2)                  ' Draw a line (in the foreground color)
                                ' from the most recent point to X2,Y2.
DO: LOOP WHILE INKEY$ = ""
CLS
LINE(0, 0)-(319, 199)           ' Draw a diagonal line across the screen.
DO: LOOP WHILE INKEY$ = ""
CLS
LINE(0, 100)-(319, 100)         ' Draw a horizontal line across the screen.
DO: LOOP WHILE INKEY$ = ""
CLS
LINE(10, 10)-(20, 20), 2        ' Draw a line in color 2.
DO: LOOP WHILE INKEY$ = ""
CLS
FOR X = 0 to 319                      ' Draw an alternating pattern
    LINE(X, 0)-(X, 199), X AND 1      ' (line on/line off) on mono-
NEXT                                  ' chrome display.
DO: LOOP WHILE INKEY$ = ""
CLS
LINE (0, 0)-(100, 100),, B      ' Draw a box in the foreground color
                                ' (note that the color is not included).
DO: LOOP WHILE INKEY$ = ""
CLS
LINE STEP(0,0)-STEP(200,200),2,BF     ' Draw a filled box in color
                                      ' 2 (coordinates are given as
                                      ' offsets with the STEP option).
DO: LOOP WHILE INKEY$ = ""
CLS
LINE(0,0)-(160,100),3,,&HFF00 ' Draw a dashed line from the
                                ' upper-left corner to the
                                ' center of the screen in color 3.
```

LINE INPUT Statement

Action Enters an entire line (up to 255 characters) from the keyboard to a string variable, without the use of delimiters.

Syntax **LINE INPUT** ⟦;⟧ ⟦"*promptstring*";⟧ *stringvariable*

Remarks The argument *promptstring* is a string constant. After *promptstring* is displayed on the screen, entry of data items into the data input field is accepted. A question mark is not printed unless it is part of *promptstring*. All input from the end of *promptstring* to the carriage return is assigned to *stringvariable*.

A semicolon immediately after the **LINE INPUT** keywords keeps the cursor at the next character position after the user presses Enter. Omitting a semicolon causes the cursor to skip to the next line.

LINE INPUT uses the same editing characters as **INPUT**.

See Also **INPUT$** Function, **INPUT** Statement

Example See the **DEF SEG** statement programming example, which uses the **LINE INPUT** statement.

LINE INPUT # Statement

Action Reads an entire line from a sequential file into a string variable.

Syntax **LINE INPUT** #*filenumber%*,*stringvariable$*

Remarks The **LINE INPUT #** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
stringvariable$	The variable the line is assigned to.

The **LINE INPUT #** statement reads all characters in the sequential file up to a carriage return. It then skips over the carriage-return-and-line-feed sequence.

LINE INPUT # is especially useful if a text file is being read one line at a time.

See Also **INPUT$** Function, **INPUT** Statement, **INPUT #** Statement, **LINE INPUT**

Example The following example creates a data file consisting of customer records that include `LAST NAME`, `FIRST NAME`, `AGE`, and `SEX`. After the file is complete, the **LINE INPUT #** statement is used to read the individual records so they can be displayed on the screen.

```
OPEN "LIST" FOR OUTPUT AS #1
PRINT "CUSTOMER INFORMATION:"
' Get customer information.
DO
    PRINT
    INPUT "   LAST NAME:  ", LName$
    INPUT "   FIRST NAME: ", FrName$
    INPUT "   AGE:        ", Age$
    INPUT "   SEX:        ", Sex$
    Sex$ = UCASE$(Sex$)
    WRITE #1, LName$, FrName$, Age$, Sex$
    INPUT "Add another"; R$
LOOP WHILE UCASE$(R$) = "Y"
CLOSE #1
```

```
' Echo the file back.
OPEN "LIST" FOR INPUT AS #1
CLS
PRINT "Records in file:": PRINT
DO WHILE NOT EOF(1)
    LINE INPUT #1, REC$    ' Read records from file.
    PRINT REC$             ' Print the records on the screen.
LOOP
' Remove file from disk.
CLOSE #1
KILL "LIST"
```

Output

```
CUSTOMER INFORMATION:

  LAST NAME:  Saintsbury
  FIRST NAME: Aloysius
  AGE:        35
  SEX:        m
Add another? y

  LAST NAME:  Frangio
  FIRST NAME: Louisa
  AGE:        27
  SEX:        f
Add another? n

Records in file:

"Saintsbury","Aloysius","35","M"
"Frangio","Louisa","27","F"
```

LOC Function

Action Returns the current position within an open file.

Syntax LOC(*filenumber%*)

Remarks The argument *filenumber%* is the number of an open file or device. With random-access files, the **LOC** function returns the number of the last record read from, or written to, the file. With sequential files, **LOC** returns the current byte position in the file, divided by 128. With binary mode files, **LOC** returns the position of the last byte read or written.

For a communications device, **LOC** returns the number of characters in the input queue waiting to be read. The value returned depends on whether the device was opened in ASCII or binary mode. In ASCII mode, the low-level routines stop queuing characters as soon as an end-of-file is received. The end-of-file itself is not queued and cannot be read. If you attempt to read the end-of-file, BASIC generates the error message `Input past end of file.` In binary mode, the end-of-file character is ignored and the entire file can be read.

For a PIPE device, **LOC** returns 1 if any data are available in the PIPE queue.

Note The **LOC** function cannot be used on ISAM tables, or the SCRN, KYBD, or LPT*n* devices.

See Also **EOF**, **LOF**, **OPEN** (File I/O)

Example The following example stops the program if the current file position is beyond 50. Note that this example is incomplete.

```
IF LOC(1) > 50 THEN STOP
```

LOCATE Statement

Action Moves the cursor to the specified position on the screen.

Syntax **LOCATE** ⟦*row%*⟧ ⟦, ⟦*column%*⟧ ⟦, ⟦*cursor%*⟧ ⟦, *start%* ⟦, *stop%*⟧⟧⟧⟧

Remarks The **LOCATE** statement uses the following arguments:

Argument	Description
row%	The number of a row on the screen; *row%* is a numeric expression returning an integer. If *row%* is not specified, the row location of the cursor does not change.
column%	The number of a column on the screen; *column%* is a numeric expression returning an integer. If *column%* is not specified, the column location of the cursor does not change.
cursor%	A Boolean value indicating whether the cursor is visible or not. A value of 0 indicates cursor *off*; a value of 1 indicates cursor *on*.
start%, *stop%*	The starting and ending raster scan lines of the cursor on the screen. The arguments *start%* and *stop%* redefine the cursor size and must be numeric expressions returning an integer between 0 and 31, inclusive.

You can omit any argument from the statement except that if *stop%* is specified, *start%* also must be specified. When you omit *row%* or *column%*, **LOCATE** leaves the cursor at the row or column where it was moved by the most recently executed input or output statement (such as **LOCATE**, **PRINT**, or **INPUT**). When you omit other arguments, BASIC assumes the previous value for the argument.

Note that the *start%* and *stop%* lines are the CRT scan lines that specify which pixels on the screen are lit. A wider range between the *start%* and *stop%* lines produces a taller cursor, such as one that occupies an entire character block. In OS/2 real mode and under DOS, **LOCATE** assumes there are eight lines (numbered 0 to 7) in the cursor. In OS/2 protected mode there are 16 lines (numbered 0 to 15).

When *start%* is greater than *stop%*, **LOCATE** produces a two-part cursor. If the *start%* line is given but the *stop%* line is omitted, *stop%* assumes the same value as *start%*. A value of 8 for both *start%* and *stop%* produces the underline cursor. The maximum cursor size is determined by the character block size of the screen mode in use. Setting *start%* greater than *stop%* displays a full-height cursor on VGA-equipped systems.

For screen mode information, see the entry for the **SCREEN** statement.

The last line on the screen is reserved for the soft-key display and is not accessible to the cursor unless the soft-key display is off (**KEY OFF**) and **LOCATE** is used with **PRINT** to write on the line.

See Also CSRLIN, POS

Examples The first example shows the effects on the cursor of different **LOCATE** statements:

```
CLS                    ' Clear screen.
LOCATE 5,5             ' Moves cursor to row 5, column 5.
PRINT "C"
DO
LOOP WHILE INKEY$ = ""
LOCATE 1,1             ' Moves cursor to upper-left corner of the screen.
PRINT "C"
DO
LOOP WHILE INKEY$ = ""
LOCATE , ,1            ' Makes cursor visible; position remains unchanged.
PRINT "C"
DO
LOOP WHILE INKEY$ = ""
LOCATE , , ,7          ' Position and cursor visibility remain unchanged;
                       ' sets the cursor to display at the bottom of the
                       ' character box starting and ending on scan line 7.
PRINT "C"
DO
LOOP WHILE INKEY$ = ""
LOCATE 5,1,1,0,7       ' Moves the cursor to line 5, column 1;
                       ' turns cursor on; cursor covers entire
                       ' character cell starting at scan line
                       ' 0 and ending on scan line 7.
PRINT "C"
DO
LOOP WHILE INKEY$ = ""
END
```

The following example prints a menu on the screen, then waits for input in the allowable range (1-4). If a number outside that range is entered, the program continues to prompt for a selection.

Note that this program is incomplete.

```
CONST FALSE = 0, TRUE = NOT FALSE
DO
   CLS
   PRINT "MAIN MENU": PRINT
   PRINT "1)  Add Records"
   PRINT "2)  Display/Update/Delete a Record"
   PRINT "3)  Print Out List of People Staying at Hotel"
   PRINT "4)  End Program"
   ' Change cursor to a block.
   LOCATE , , 1, 1, 12
   LOCATE 12, 1
   PRINT "What is your selection?";
   DO
      CH$ = INPUT$(1)
   LOOP WHILE (CH$ < "1" OR CH$ > "4")
   PRINT CH$

' Call the appropriate SUB procedure.
   SELECT CASE VAL(CH$)
      CASE 1
         CALL Add
      CASE 2
         CALL Search
      CASE 3
         CALL Hotel
      CASE 4
         CALL Quit
   END SELECT
LOOP WHILE NOT ENDPROG
  .
  .
  .
END
```

LOCK...UNLOCK Statement

Action Controls access by other processes to all or part of an opened file.

Syntax **LOCK** [[#]]*filenumber%* [[,{*record&* | [[*start&*]] **TO** *end&*}]]

.
.
.

UNLOCK [[#]]*filenumber%* [[,{*record&* | [[*start&*]] **TO** *end&*}]]

Remarks These statements are used in networked environments where several processes might need access to the same file. The **LOCK** and **UNLOCK** statements use the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
record&	The number of the record or byte to be locked. The argument *record&* can be any number from 1 to 2,147,483,647 (equivalent to $2^{31}-1$). A record can be up to 32,767 bytes in length.
start&	The number of the first record or byte to be locked.
end&	The number of the last record or byte to be locked.

For binary-mode files, *record&*, *start&*, and *end&* represent the number of a byte relative to the beginning of the file. The first byte in a file is byte 1.

For random-access files, *record&*, *start&*, and *end&* represent the number of a record relative to the beginning of the file. The first record is record 1.

If the file has been opened for sequential input or output, **LOCK** and **UNLOCK** affect the entire file, regardless of the range specified by *start&* and *end&*.

The **LOCK** and **UNLOCK** statements are always used in pairs. The arguments to **LOCK** and **UNLOCK** must match exactly.

If you specify just one record, then only that record is locked or unlocked. If you specify a range of records and omit a starting record (*start&*), then all records from the first record to the end of the range (*end&*) are locked or unlocked. **LOCK** with no *record&* locks the entire file, while **UNLOCK** with no *record&* unlocks the entire file.

LOCK and **UNLOCK** execute only at run time if you are using OS/2 or versions of DOS that support networking (version 3.1 or later). In DOS, you must run the SHARE.EXE program to enable locking operations. Using earlier versions of DOS causes BASIC to generate the error message `Feature unavailable` if **LOCK** and **UNLOCK** are executed.

Warning

Be sure to remove all locks with an **UNLOCK** statement before closing a file or terminating your program. Failing to remove locks produces unpredictable results.

The arguments to **LOCK** and **UNLOCK** must match exactly.

Do not use **LOCK** and **UNLOCK** on devices or ISAM tables.

If you attempt to access a file that is locked, BASIC may generate the following error messages:

```
Bad record number
```

```
Permission denied
```

Example

The following example illustrates the use of the **LOCK** and **UNLOCK** statements. The program creates a sample data record in a random-access file. Once the sample record is created and closed, the program again opens the file to allow customer information to be updated. As a record is opened, it is locked to prevent use by another terminal that has access to the same file. After the update is complete, the record is unlocked, again allowing modifications by others with access.

```
' Define the record.
TYPE AccountRec
   Payer AS STRING * 20
   Address AS STRING * 20
   Place AS STRING * 20
   Owe AS SINGLE
END TYPE

DIM CustRec AS AccountRec

' This section creates a sample record to use.
ON ERROR GOTO ErrHandler

OPEN "MONITOR" FOR RANDOM SHARED AS #1 LEN = LEN(CustRec)

CustRec.Payer = "George Washington"
CustRec.Address = "1 Cherry Tree Lane"
CustRec.Place = "Mt. Vernon, VA"
CustRec.Owe = 12!
PUT #1, 1, CustRec            ' Put one record in the file.
CLOSE #1
```

```
' This section opens the sample record for updating.
OPEN "MONITOR" FOR RANDOM SHARED AS #1 LEN = LEN(CustRec)
DO
    Number% = 0              ' Reset to zero.
    DO UNTIL Number% = 1     ' Force user to input 1.
        CLS : LOCATE 10, 10
        INPUT "Customer Number?   #"; Number%
    LOOP
    ' Lock the current record so another process
    ' doesn't change it while you're using it.
    LOCK #1, Number%
    GET #1, Number%, CustRec
    LOCATE 11, 10: PRINT "Customer: "; CustRec.Payer
    LOCATE 12, 10: PRINT "Address:  "; CustRec.Address
    LOCATE 13, 10: PRINT "Currently owes: $"; CustRec.Owe
    LOCATE 15, 10: INPUT "Change (+ or -)", Change!
    CustRec.Owe = CustRec.Owe + Change!
    PUT #1, Number%, CustRec
    ' Unlock the record.
    UNLOCK #1, Number%
    LOCATE 17, 10: INPUT "Update another? ", Continue$
    Update$ = UCASE$(LEFT$(Continue$, 1))
LOOP WHILE Update$ = "Y"
CLOSE #1
KILL "MONITOR"  ' Remove file from disk.
END

ErrHandler:
IF ERR = 70 THEN        ' Permission-denied error.
    CLS
    PRINT "You must run SHARE.EXE before running this example."
    PRINT "Exit the programming environment, run SHARE.EXE, and"
    PRINT "reenter the programming environment to run this"
    PRINT "example. Do not shell to DOS to run SHARE.EXE or you"
    PRINT "may not be able to run other programs until you reboot."
END IF
END
```

LOF Function

Action Returns the size of an open file (in bytes), the number of records in an ISAM table, or, when used with the **OPEN COM** statement, the number of bytes free in the output buffer.

Syntax **LOF**(*filenumber%*)

Remarks The argument *filenumber%* is the number used in the **OPEN** statement to open the file, device, or ISAM table.

Important **LOF** can be used only on disk files, ISAM tables, or COM devices. It cannot be used with the BASIC devices SCRN, KYBD, CONS, LPT*n*, or PIPE.

See Also **BOF**, **EOF**, **LOC**, **OPEN** (File I/O)

Example See the programming example for the **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements, which uses the **LOF** function in the context of ISAM.

LOG Function

Action Returns the natural logarithm of a numeric expression.

Syntax **LOG**(*numeric-expression*)

Remarks The argument *numeric-expression* must be greater than zero. The natural logarithm is the logarithm to the base *e*. The constant *e* is approximately equal to 2.718282.

LOG is calculated in single precision if *numeric-expression* is an integer or single-precision value. If you use any other numeric type, **LOG** is calculated in double precision.

You can calculate base-10 logarithms by dividing the natural logarithm of the number by the natural logarithm of 10. The following **FUNCTION** procedure calculates base-10 logarithms:

```
FUNCTION Log10(X) STATIC
   Log10=LOG(X)/LOG(10#)
END FUNCTION
```

See Also **EXP**

Example The following example prints the value of *e* and then prints the natural logarithms of *e* taken to the first, second, and third powers:

```
CLS                        ' Clear screen.
PRINT EXP(1),
FOR I = 1 TO 3
    PRINT LOG(EXP(1) ^ I),
NEXT
PRINT
```

Output

```
2.718282       1       2       3
```

LPOS Function

Action Returns the number of characters sent to the printer since the last carriage return was sent.

Syntax **LPOS**(*n%*)

Remarks The argument *n%* is an integer expression with a value between 0 and 3 that indicates one of the printers. For example, LPT1 would be tested with `LPOS(1)` or `LPOS(0)`, LPT2 would be tested with `LPOS(2)`, and LPT3 would be tested with `LPOS(3)`.

The **LPOS** function does not necessarily give the physical position of the print head because it does not expand tab characters. In addition, some printers may buffer characters.

Example See the **LPRINT** statement programming example, which uses the **LPOS** function.

LPRINT, LPRINT USING Statements

Action Print data on the printer LPT1.

Syntax **LPRINT** [[*expressionlist*]] [[{;|,}]]
LPRINT USING *formatstring$*; *expressionlist* [[{;|,}]]

Remarks These statements function in the same way as the **PRINT** and **PRINT USING** statements except that output goes to the printer.

The following list describes the parts of the **LPRINT** and **LPRINT USING** statements:

Part	Description
expressionlist	The values that are printed on printer LPT1.
formatstring$	Specifies the format, using the same formatting characters as the **PRINT USING** statement.
{;|,}	Determines the location on the page of the first value printed by the next statement to use LPT1 (such as the next **LPRINT** statement or a **PRINT #** or **WRITE #** directing data to LPT1). The semicolon means to print immediately after the last value in this **LPRINT** statement; the comma means to print at the start of the next unoccupied print zone.

The printer output from an **LPRINT** statement will be the same as the screen output from a **PRINT** statement, if both statements have the same *expressionlist* values and output-line width.

The printer output from an **LPRINT USING** statement will be the same as the screen output from a **PRINT USING** statement, if both statements have the same values for *formatstring$*, *expressionlist*, and output-line width.

The **LPRINT** statement assumes an 80-character-wide printer. This width can be changed with a **WIDTH** statement.

If you use **LPRINT** with no arguments, a blank line is printed.

Warning Because the **LPRINT** statement uses the LPT1 printer device, you should not use **LPRINT** in a program that also contains an `OPEN "LPT1"` statement. Using these two statements together produces unpredictable results.

BASICA An `LPRINT CHR$(13)` statement actually outputs both `CHR$(13)` and `CHR$(10)`. This feature was created to provide compatibility with BASICA.

See Also PRINT, PRINT USING, WIDTH

Example The following example prompts the user for team names and the names of players on each team. Then it prints the players and their teams on the printer.

```
CLS      ' Clear screen.
LPRINT "Team Members"; TAB(76); "TEAM" : LPRINT
INPUT "How many teams"; TEAMS
INPUT "How many players per team";PPT
PRINT
FOR T = 1 TO TEAMS
   INPUT "Team name: ", TEAM$
   FOR P = 1 TO PPT
      INPUT "   Enter player name: ", PLAYER$
      LPRINT PLAYER$;
      IF P < PPT THEN
         IF LPOS(0) > 55 THEN                ' Print a new line if print
            LPRINT : LPRINT "     ";         ' head is past column 55.
         ELSE
            LPRINT ", ";                     ' Otherwise, print a comma.
         END IF
      END IF
   NEXT P
   LPRINT STRING$(80-LPOS(0)-LEN(TEAM$),"."); TEAM$
NEXT T
```

LSET Statement

Action Moves data from memory to a random-access file buffer (in preparation for a **PUT** statement), copies one record variable to another, or left-justifies the value of a string in a string variable.

Syntax 1 **LSET** *stringvariable$=stringexpression$*

Syntax 2 **LSET** *recordvariable1=recordvariable2*

Remarks The **LSET** statement uses the following arguments:

Argument	Description
stringvariable$	Usually a random-access file field defined in a **FIELD** statement, although it can be any string variable.
stringexpression$	The value that is assigned to *stringvariable$* and is left-justified.
recordvariable1	A record variable of any data type.
recordvariable2	A record variable of any data type.

If *stringexpression$* requires fewer bytes than were defined for *stringvariable$* in the **FIELD** statement, the **LSET** function left-justifies the string in the field (**RSET** right-justifies the string). Spaces are used to pad the extra positions. If the string is too long for the field, both **LSET** and **RSET** truncate characters from the right. Numeric values must be converted to strings before they are justified with the **LSET** or **RSET** statements.

You also can use **LSET** or **RSET** with a string variable not defined in a **FIELD** statement to left-justify or right-justify a string in a given field. For example, the following program lines will right-justify the string `N$` in a 20-character field:

```
A$=SPACE$(20)
RSET A$=N$
```

This can be useful for formatting printed output.

You can use **LSET** with Syntax 2 to assign one record variable to another. The following example copies the contents of `RecTwo` to `RecOne`:

```
TYPE TwoString
   StrFld AS STRING * 2
END TYPE

TYPE ThreeString
   StrFld AS STRING * 3
END TYPE
DIM RecOne AS TwoString, RecTwo AS ThreeString
.
.
.
LSET RecOne = RecTwo
```

Notice that **LSET** is used to assign record variables of differing types. Record variables of the same type also can be assigned using **LET**. Also, because `RecOne` is only two bytes long, only two bytes are copied from `RecTwo`. **LSET** copies only the number of bytes in the shorter of the two record variables.

See Also **LET**; **MKI$**, **MKL$**, **MKS$**, **MKD$**, **MKC$**; **PUT** (File I/O); **RSET**

Example The following example shows the effects of using **LSET** and **RSET** to assign values to fixed- and variable-length strings:

```
DIM TmpStr2 AS STRING * 25
CLS    ' Clear screen.

' Use RSET on variable-length string with a value.
TmpStr$ = SPACE$(40)
PRINT "Here are two strings that have been right and left "
PRINT "justified in a 40-character variable-length string."
PRINT

RSET TmpStr$ = "Right-|"
PRINT TmpStr$
LSET TmpStr$ = "|-Left"
PRINT TmpStr$
PRINT "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^"
PRINT "12345678901234567890123456789012345678901234567890"
PRINT "         1         2         3         4         5"
```

```
' Use RSET on fixed-length string of length 25.
PRINT
PRINT
PRINT "Here are two strings that have been right and left"
PRINT "justified in a 25-character fixed-length string."
PRINT
RSET TmpStr2 = "Right-|"
PRINT TmpStr2
LSET TmpStr2$ = "|-Left"
PRINT TmpStr2$
PRINT "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^"
PRINT "12345678901234567890123456789012345678901234567890"
PRINT "         1         2         3         4         5"
```

Output

```
Here are two strings that have been right and left
justified in a 40-character variable-length string.

                                 Right-|
|-Left
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
12345678901234567890123456789012345678901234567890
         1         2         3         4         5


Here are two strings that have been right and left
justified in a 25-character fixed-length string.

                  Right-|
|-Left
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
12345678901234567890123456789012345678901234567890
         1         2         3         4         5
```

LTRIM$ Function

Action Returns a copy of a string with leading spaces removed.

Syntax **LTRIM$**(*stringexpression$*)

Remarks The *stringexpression$* can be any string expression. The **LTRIM$** function works with both fixed- and variable-length string variables.

See Also **RTRIM$**

Example The following example copies a file to a new file, removing all leading and trailing spaces:

```
' Get the filenames.
INPUT "Enter input filename:", InFile$
INPUT "Enter output filename:", OutFile$

OPEN InFile$ FOR INPUT AS #1
OPEN OutFile$ FOR OUTPUT AS #2

' Read, trim, and write each line.
DO WHILE NOT EOF(1)
   LINE INPUT #1, LineIn$
   ' Remove leading and trailing blanks.
   LineIn$ = LTRIM$(RTRIM$(LineIn$))
   PRINT #2, LineIn$
LOOP

CLOSE #1, #2
END
```

MID$ Function

Action Returns a substring of a string.

Syntax **MID$**(*stringexpression$*,*start%*[[,*length%*]])

Remarks The **MID$** function uses the following arguments:

Argument	Description
stringexpression$	The string expression from which the substring is extracted. This can be any string expression.
start%	The character position in *stringexpression$* where the substring starts.
length%	The number of characters to extract.

The arguments *start%* and *length%* must be between 1 and 32,767, inclusive. If *length%* is omitted or if there are fewer than *length%* characters in the string (including the *start%* character), the **MID$** function returns all characters from the position *start%* to the end of the string.

If *start%* is greater than the number of characters in *stringexpression$*, **MID$** returns a null string.

Use the **LEN** function to find the number of characters in *stringexpression$*.

See Also **LEFT$**, **LEN**, **MID$** Statement, **RIGHT$**

Example The following example converts a binary number to a decimal number. The program uses the **MID$** function to extract digits from the binary number (input as a string).

```
INPUT "Binary number = ", Binary$    ' Input binary number as string.
Length = LEN(Binary$)                ' Get length of string.
Decimal = 0
FOR K = 1 TO Length
   Digit$ = MID$(Binary$, K, 1)           ' Get digit from string.
   IF Digit$ = "0" OR Digit$ = "1" THEN   ' Test for binary digit.
      Decimal = 2 * Decimal + VAL(Digit$) ' Convert digits to numbers.
   ELSE
      PRINT "Error--invalid binary digit: "; Digit$
      EXIT FOR
   END IF
NEXT
PRINT "Decimal number =" Decimal
```

MID$ Statement

Action Replaces a portion of a string variable with another string.

Syntax **MID$**(*stringvariable$*,*start%* [[,*length%*]])=*stringexpression$*

Remarks The **MID$** statement uses the following arguments:

Argument	Description
stringvariable$	The string variable being modified.
start%	A numeric expression giving the position in *stringvariable$* where the replacement starts.
length%	A numeric expression that gives the length of the string being replaced.
stringexpression$	The string expression that replaces part of *stringvariable$*.

The arguments *start%* and *length%* are integer expressions. The argument *stringvariable$* is a string variable, but *stringexpression$* can be a string variable, a string constant, or a string expression.

The optional *length%* refers to the number of characters from the argument *stringexpression$* that are used in the replacement. If *length%* is omitted, all of *stringexpression$* is used. However, regardless of whether *length%* is omitted or included, the replacement of characters never goes beyond the original length of *stringvariable$*.

See Also **MID$** Function

Example This example uses the **MID$** statement to replace string characters:

```
CLS                              ' Clear screen.
Test$ = "Paris, France"
PRINT Test$
MID$(Test$, 8)="Texas "          ' Starting at position 8, replace
                                 ' characters in Test$ with Texas.
PRINT Test$
```

Output

```
Paris, France
Paris, Texas
```

MKDIR Statement

Action Creates a new directory.

Syntax **MKDIR** *pathname$*

Remarks The *pathname$* is a string expression that specifies the name of the directory to be created in DOS. The *pathname$* must be a string of fewer than 64 characters.

The **MKDIR** statement works like the DOS command MKDIR. However, the syntax in BASIC cannot be shortened to MD, as it can in DOS.

You can use **MKDIR** to create a DOS directory with a name that contains an embedded space. However, although you can access that directory through DOS, you can remove it only with the BASIC **RMDIR** statement.

See Also **CHDIR**, **RMDIR**

Example See the **CHDIR** statement programming example, which uses the **MKDIR** statement.

MKI$, MKL$, MKS$, MKD$, and MKC$ Functions

Action Convert numeric values to string values.

Syntax **MKI$**(*integer-expression%*)
MKL$(*long-integer-expression&*)
MKS$(*single-precision-expression!*)
MKD$(*double-precision-expression#*)
MKC$(*currency-expression@*)

Remarks **MKI$**, **MKL$**, **MKS$**, **MKD$** and **MKC$** are used with **FIELD** and **PUT** statements to write numbers to a random-access file. The functions convert numeric expressions to strings that can be stored in the strings defined in the **FIELD** statement. These functions are the inverse of **CVI**, **CVL**, **CVS**, **CVD**, and **CVC**.

The following table describes these numeric-conversion functions:

Function	Description
MKI$	Converts an integer to a 2-byte string.
MKL$	Converts a long-integer value to a 4-byte string.
MKS$	Converts a single-precision value to a 4-byte string.
MKD$	Converts a double-precision value to an 8-byte string.
MKC$	Converts a currency value to an 8-byte string.

Note These BASIC record variables provide a more efficient and convenient way of reading and writing random-access files than some older versions of BASIC.

See Also **CVI**, **CVL**, **CVS**, **CVD**, **CVC**; **GET** (File I/O); **FIELD**; **PUT** (File I/O)

Example See the programming example for the **CVI**, **CVL**, **CVS**, **CVD**, and **CVC** statements, which uses the **MKS$** function.

MKSMBF$, MKDMBF$ Functions

Action Convert an IEEE-format number to a string containing a Microsoft-Binary-format number.

Syntax **MKSMBF$**(*single-precision-expression!*)
MKDMBF$(*double-precision-expression#*)

Remarks These functions are used to write real numbers to random-access files using Microsoft Binary format. They are particularly useful for maintaining data files created with older versions of BASIC.

The **MKSMBF$** and **MKDMBF$** functions convert real numbers in IEEE format to strings so they can be written to the random-access file.

To write a real number to a random-access file in Microsoft Binary format, convert the number to a string using **MKSMBF$** (for a single-precision number) or **MKDMBF$** (for a double-precision number). Then store the result in the corresponding field (defined in the **FIELD** statement) and write the record to the file using the **PUT** statement.

Currency numbers are not real numbers, so they cannot be converted to strings that contain Microsoft-Binary-format numbers.

See Also **CVSMBF, CVDMBF; CVI, CVL, CVS, CVD, CVC; MKI$, MKL$, MKS$, MKD$, MKC$**

Example The following example stores real values in a file as Microsoft-Binary-Format numbers:

```
TYPE Buffer
   NameField AS STRING * 20
   ScoreField AS STRING * 4
END TYPE
DIM RecBuffer AS Buffer
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 LEN=LEN(RecBuffer)
PRINT "Enter a name and a score, separated by a comma."
PRINT "Enter 'END, 0' to end input."
INPUT NameIn$, Score
I=0   ' Read names and scores from the console until the name is END.
DO WHILE UCASE$(NameIn$) <> "END"
   I=I+1
   RecBuffer.NameField=NameIn$
   RecBuffer.ScoreField=MKSMBF$(Score)    ' Convert to a string.
   PUT #1,I,RecBuffer
   INPUT NameIn$, Score
LOOP
PRINT I;" records written."
CLOSE #1
```

MOVEFIRST, MOVELAST, MOVENEXT, MOVEPREVIOUS Statements

Action Make records current according to their relative position in an ISAM table.

Syntax
MOVEFIRST [[#]]*filenumber%*
MOVELAST [[#]]*filenumber%*
MOVENEXT [[#]]*filenumber%*
MOVEPREVIOUS [[#]]*filenumber%*

Remarks The *filenumber%* is the number used in the **OPEN** statement to open the table. The **MOVEFIRST**, **MOVELAST**, **MOVENEXT**, and **MOVEPREVIOUS** statements allow you to make records current according to their relative position on the current index in an ISAM table:

Statement	*Effect*
MOVEFIRST	First record becomes current.
MOVELAST	Last record becomes current.
MOVENEXT	Next record becomes current.
MOVEPREVIOUS	Previous record becomes current.

The action of these statements is made relative to the record that is current according to the current index.

Using **MOVEPREVIOUS** when the first indexed record is current moves the current position to the beginning of the table. The beginning of an ISAM table is the position before the first record according to the current index.

Using **MOVENEXT** when the last indexed record is current moves the current position to the end of the table. The end of an ISAM table is the position following the last record according to the current index.

See Also **BOF**; **EOF**; **SEEKGT**, **SEEKGE**, **SEEKEQ**

Example See the **CREATEINDEX** statement programming example, which uses the **MOVEFIRST**, **MOVELAST**, **MOVENEXT**, and **MOVEPREVIOUS** statements.

NAME Statement

Action Changes the name of a disk file or directory.

Syntax **NAME** *oldfilespec$* **AS** *newfilespec$*

Remarks The **NAME** statement is similar to the DOS RENAME command. **NAME** can move a file from one directory to another but cannot move a directory.

The arguments *oldfilespec$* and *newfilespec$* are string expressions, each of which contains a filename and an optional path. If the path in *newfilespec$* is different from the path in *oldfilespec$*, the **NAME** statement renames the file as indicated.

The file *oldfilespec$* must exist and *newfilespec$* must not be in use. Both files must be on the same drive. If you use **NAME** with different drive designations in the old and new filenames, BASIC generates the error message `Rename across disks.`

Using **NAME** on an open file causes BASIC to generate the run-time error message `File already open.` You must close an open file before renaming it.

Example The following example shows how to use **NAME** to rename a README.DOC file:

```
CLS    ' Clear screen.
'Change README.DOC to READMINE.DOC
NAME "README.DOC" AS "READMINE.DOC"
FILES  ' Display the files in your directory to confirm name change.
PRINT
PRINT "README.DOC is now called READMINE.DOC"
PRINT "Press any key to continue"
DO
LOOP WHILE INKEY$ = ""
' Change name back to README.DOC and confirm by displaying files.
NAME "READMINE.DOC" AS "README.DOC"
FILES
PRINT
PRINT "README.DOC is back to its original name"
```

You also can use **NAME** to move a file from one directory to another. You can move README.DOC to the directory `\MYDIR` with the following statement:

```
NAME "README.DOC" AS "\MYDIR\README.DOC"
```

OCT$ Function

Action Returns a string representing the octal value of the numeric argument.

Syntax **OCT$**(*numeric-expression*)

Remarks The argument *numeric-expression* can be of any type. It is rounded to an integer or long integer before the **OCT$** function evaluates it.

See Also **HEX$**

Example The following example displays the octal version of several decimal numbers:

```
PRINT "The octal representation of decimal 24 is " OCT$(24)
PRINT "The octal representation of decimal 55 is " OCT$(55)
PRINT "The octal representation of decimal 101 is " OCT$(101)
```

Output

```
The octal representation of decimal 24 is 30
The octal representation of decimal 55 is 67
The octal representation of decimal 101 is 145
```

ON ERROR Statement

Action

Enables an error-handling routine and specifies the location of that routine in the program. The **ON ERROR** statement also can be used to disable an error-handling routine.

Syntax

ON ⟦**LOCAL**⟧ **ERROR** {**GOTO** *line* | **RESUME NEXT** | **GOTO 0**}

Remarks

If no **ON ERROR** statement is used in a program, any run-time error will be fatal (BASIC generates an error message and program execution stops).

The following list describes the parts of the **ON ERROR** statement:

Part	Description
LOCAL	Used to indicate an error-handling routine in the same procedure. If not used, module-level error handlers are indicated.
GOTO *line*	Enables the error-handling routine that starts at *line*. Thereafter, if a run-time error occurs, program control branches to *line* (a label or a line number). The specified line is either in the module-level code or in the same procedure (if the **LOCAL** keyword is used). If not found in either place, BASIC generates a `Label not defined` compile-time error.
RESUME NEXT	Specifies that, when a run-time error occurs, control goes to the statement after the statement where the error occurred; the **ERR** function then can be used to obtain the run-time error code.
GOTO 0	Disables any enabled module-level error-handling routine within the current module, or disables any enabled error handler within the current procedure (if used together with the **LOCAL** keyword).

The **LOCAL** keyword indicates an error-handling routine that is "local" to the procedure within which the error-handling routine is located. A local error-handling routine:

- Overrides any enabled module-level error-handling routines.
- Is enabled only while the procedure within which it is located is executing.

Notice that the local error handler remains enabled while any procedures execute that are directly or indirectly called by the procedure within which the error handler is located. One of those procedures may supersede the local error handler.

Note The module-level error-handling routines provided by previous versions of BASIC may be all you need; you may never need to use the **LOCAL** keyword.

Use of the **RESUME NEXT** option causes program execution to resume with the statement immediately following the statement that caused the run-time error. This enables your program to continue execution of a block of program code, despite a run-time error, then check for the cause of the error. This also enables you to build the error-handling code in line with the main module code or procedure, rather than at a remote location in the program.

The statement form **ON ERROR GOTO 0** disables error handling. It does not specify line 0 as the start of the error-handling code, even if the program or procedure contains a line numbered 0.

Notice that an error-handling routine is not a **SUB** or **FUNCTION** procedure or a **DEF FN** function. An error-handling routine is a block of code marked by a line label or line number.

An error handler is enabled when it is referred to by an **ON ERROR GOTO** *line* statement. Once an error handler is enabled, a run-time error causes program control to jump to the enabled error-handling routine and makes the error handler "active." An error handler is active from the time a run-time error has been trapped until a **RESUME** statement is executed in the handler.

Error-handling routines must rely on the value in **ERR** to determine the cause of the error. The error-handling routine should test or save this value before any other error can occur or before calling a procedure that could cause an error. The value in **ERR** reflects only the last error to occur.

If you use the BASIC command line to compile a program that has been developed in the QBX environment and has error-handling routines, compile with the /E or /X option. The Make EXE File command in the QBX environment uses these options.

If an error occurs in a procedure or module that does not have an enabled error-handling routine, BASIC searches for an enabled, inactive error handler to trap the error. BASIC searches back through all the procedures and modules that have called the procedure where the error occurred.

Using Module-Level Error Handlers

Once enabled by execution of an **ON ERROR GOTO** *line* statement, a module-level error handler stays enabled unless explicitly disabled by execution of an **ON ERROR GOTO 0** statement somewhere in the module.

In the following cases, BASIC searches back through the module-level code of the calling modules, looking for an enabled, inactive error handler:

- In a multiple-module program that contains only module-level error handlers, if an enabled and inactive error handler cannot be found in the module where the error occurred.
- If an error occurs in a module-level error handler itself (BASIC does not automatically treat this as a fatal error).
- If an **ON ERROR GOTO 0** statement is encountered while the current module's module-level error handler is active.

Procedure-Level Error Handlers

SUB and **FUNCTION** procedures and **DEF FN** functions can contain their own error-handling routines.

To enable a local error handler, use the statement **ON LOCAL ERROR GOTO** *line*. The argument *line* must be a label or line number in the same procedure as the **ON LOCAL ERROR GOTO** statement.

The local error handler is automatically disabled when the procedure returns, or by execution of the statement **ON LOCAL ERROR GOTO 0**.

You will want program flow to avoid the statements that make up the error-handling routine. One way to keep the error-handling code from executing when there is no error is to place an **EXIT SUB**, **EXIT FUNCTION**, or **EXIT DEF** statement immediately ahead of the error-handling routine, as in the following example:

```
SUB InitializeMatrix (var1, var2, var3, var4)
      .
      .
      .
   ON LOCAL ERROR GOTO ErrorHandler
      .
      .
      .
   EXIT SUB
ErrorHandler:
      .
      .
      .
   RESUME NEXT
END SUB
```

The example shows the error-handling code located after the **EXIT SUB** statement and before the **END SUB** statement. This partitions the error-handling code from the normal execution flow of the procedure. However, error-handling code can be placed anywhere in a procedure.

In the QBX environment, or if the command-line compiler is used with the /O option:

- If a local error handler is active and an **END SUB**, **END FUNCTION**, or **END DEF** statement is encountered, BASIC generates the run-time error message `No RESUME`.
- If an **EXIT SUB**, **EXIT FUNCTION**, or **EXIT DEF** statement is encountered, BASIC does not generate a run-time error; in other words, it is assumed that this does not represent a logic error in the program.

In a multiple-module program that contains only procedure-level (local) error handlers, if an enabled and inactive error handler cannot be found in the procedure where the error occurred, BASIC searches back through all the calling procedures as well as the module-level code of all the calling modules, looking for an enabled, inactive error handler.

Using Both Module-Level and Procedure-Level Error Handlers

For simplicity and clarity, avoid using both local- and module-level error handlers in the same program, except for using an error handler in the main module module-level code as the last line of defense against a fatal error. If you need to, however, you can have both a module-level error handler and a local error handler enabled at the same time. (In fact, you can enable both local- and module-level error handling from within a procedure.)

While searching for an enabled, inactive error handler, BASIC may encounter an active event handler. Unless an enabled error-handling routine is provided in the same module-level code as the event handler, BASIC generates a fatal error. Therefore, to produce completely bullet-proof BASIC code in programs that trap events, make sure an error handler in the same module-level code as the event handler is enabled at the time the event occurs. Because event handlers can be located only at the module level, this is another special case where you might mix module-level and procedure-level error handlers.

Notice that when an error-handling routine is finished and program execution resumes through execution of a **RESUME 0** or **RESUME NEXT** statement, the resume location is based on the location of the error-handling routine, and not necessarily on the location where the error occurred. For more information, see the entry for **RESUME**.

See Also **ERL, ERR; ERROR; RESUME**

Example The following program prompts the user for a disk drive designation. Once the user has input the drive designation, the program attempts to write a large file to the disk. A number of errors can occur.

```
DECLARE SUB ErrorMessage (Message$)
DECLARE SUB WriteBigFile (Filenum%)

ON ERROR GOTO ErrHandler
CLS
PRINT "This program will attempt to write a large file to a disk drive that"
PRINT "you have selected. The file will be erased when the program ends."
PRINT
DO
   INPUT "Enter the letter of the drive where you want this file written";
DR$
   DR$ = UCASE$(DR$)
LOOP UNTIL LEN(DR$) >= 1 AND LEN(DR$) <= 2 AND DR$ >= "A" AND DR$ <= "Z"
IF LEN(DR$) > 1 THEN
   IF RIGHT$(DR$, 1) <> ":" THEN
   DR$ = LEFT$(DR$, 1) + ":"
   END IF
ELSE
   DR$ = DR$ + ":"
END IF
```

```
' Put together a complete file specification.
FileSpec$ = DR$ + "BIGFILE.XXX"
' Get the next available file number.
Filenum% = FREEFILE
' Try an open the file
OPEN FileSpec$ FOR OUTPUT AS Filenum%
WriteBigFile Filenum%
CLOSE Filenum%
CLS
PRINT "Everything was OK. No errors occurred."
PRINT "Deleting the file that was created."
KILL FileSpec$
' Same as END, returns to operating system.
SYSTEM

ErrHandler:
SELECT CASE ERR
   CASE 52'Bad file name or number.
   END
   CASE 53' File not found.
   RESUME NEXT
   CASE 57' Device I/O error.
   ErrorMessage "You should probably format the disk."
   END
   CASE 64' Bad file name.
   ErrorMessage "The drive name you specified was not correct."
   END
   CASE 68' Device unavailable
   ErrorMessage "The drive you named is unavailable."
   END
   CASE 71' Drive not ready
   ErrorMessage "The drive was not ready. Check the drive!"
   END
   CASE ELSE
   ErrorMessage "An unexpected FATAL error has occurred."
   STOP
END SELECT
```

```
SUB ErrorMessage (Message$)
   ON LOCAL ERROR GOTO MessageError
   CLS
   PRINT Message$
   PRINT "Cannot continue."
   PRINT
   PRINT "Press any key to exit."
   DO
   LOOP WHILE INKEY$ = ""
   EXIT SUB
MessageError:
   RESUME NEXT
END SUB

SUB WriteBigFile (Filenum%)
   ON LOCAL ERROR GOTO LocalHandler
   TEXT$ = STRING$(1024, "A")
   FOR I% = 1 TO 400
      PRINT #Filenum%, TEXT$
   NEXT I%
   EXIT SUB
LocalHandler:
   SELECT CASE ERR
   CASE 61 ' Disk full
      ErrorMessage ("There is no room remaining on the disk.")
      KILL "BIGFILE.XXX"
      END
   CASE ELSE
      ERROR ERR
   END SELECT
END SUB
```

ON event Statements

Action Indicate the first line of an event-trapping routine.

Syntax **ON** *event* **GOSUB** {*linenumber* | *linelabel*}

Remarks The **ON** *event* statements let you specify a routine that is executed whenever a specified type of event occurs on a specified device. The **ON** *event* statements use the following arguments:

Argument	Description
event	Specifies the event that causes a branch to the event-trapping routine. See below for more information about specific types of events.
linenumber or *linelabel*	The number or label of the first line in the event-trapping routine. This line must be in the module-level code. A *linenumber* value of 0 disables event trapping and does not specify line 0 as the start of the routine.

The following list describes the events that can be trapped:

Event	Description
COM(*n%*)	Branches to the routine when characters are received at a communications port. The integer expression *n%* indicates one of the serial ports, either 1 or 2. For more information, see "Using ON COM" later in this entry.
KEY(*n%*)	Branches to the routine when a specified key is pressed. The integer expression *n%* is the number of a function key, direction key, or user-defined key. For more information, see "Using ON KEY" later in this entry.
PEN	Branches to the routine when a lightpen is activated. For more information, see "Using ON PEN" later in this entry.
PLAY(*queuelimit%*)	Branches when there are fewer than *queuelimit%* notes in the background-music queue. A **PLAY** event-trapping routine can be used with a **PLAY** statement to play music in the background while a program is running. For more information, see "Using ON PLAY" later in this entry.
SIGNAL(*n%*)	Branches to the routine when an OS/2 protected-mode signal is received. For more information, see "Using ON SIGNAL" later in this entry.

STRIG(*n%*)	Branches to the event-trapping routine when a joystick trigger is pressed. The integer expression *n%* is the joystick trigger number. For more information, see "Using ON STRIG" later in this entry.
TIMER(*n&*)	Branches to the routine when *n* seconds have passed. The numeric expression *n&* is in the range 1 to 86,400 (1 second to 24 hours).
UEVENT	Branches to the event-trapping routine for a user-defined event. This event usually is a hardware interrupt (although it can be a software interrupt). For more information, see "Using ON UEVENT" later in this entry.

The **ON** *event* statement specifies only the start of an event-trapping routine. Another set of statements determines whether or not the routine is called. This set of statements turns event trapping on or off and determines how events are handled when trapping is off. The following list describes these statements in a general way. See the entries for particular statements for more specific information.

Event	Description
event **ON**	Enables event trapping. Event trapping occurs only after an *event* **ON** statement is executed.
event **OFF**	Disables event trapping. No trapping takes place until the execution of another *event* **ON** statement. Events occurring while trapping is off are ignored.
event **STOP**	Suspends event trapping so no trapping takes place until an *event* **ON** statement is executed. Events occurring while trapping is inhibited are remembered and processed when an *event* **ON** statement is executed.

When an event trap occurs and the routine is called, BASIC performs an automatic *event* **STOP** that prevents recursive traps. The **RETURN** statement from the trapping routine automatically performs an *event* **ON** statement unless an explicit *event* **OFF** is performed inside the routine.

If your program contains event-handling statements and you are compiling from the BC command line, use the BC /W or /V option. (The /W option checks for events at every label or line number; the /V option checks at every statement.) If you do not use these options and your program contains event traps, BASIC generates the error message `ON events without /V or /W on command line.`

Note Because of the implicit *event* **STOP** and *event* **ON** statements, events that occur during execution of the trapping routine are remembered and processed when the routine ends.

The **RETURN** *linenumber* or **RETURN** *linelabel* forms of **RETURN** can be used to return to a specific line from the trapping routine. Use this type of return with care, however, because any **GOSUB**, **WHILE**, or **FOR** statements active at the time of the trap remain active. BASIC may generate error messages such as `NEXT without FOR`. In addition, if an event occurs in a procedure, a **RETURN** *linenumber* or **RETURN** *linelabel* statement cannot get back into the procedure because the line number or label must be in the module-level code.

The next sections contain additional information about the **ON COM**, **ON KEY**, **ON PEN**, **ON PLAY**, **ON SIGNAL**, **ON STRIG**, and **ON UEVENT** statements.

Using ON COM

You can use **ON COM**(*n%*) to specify a routine to branch to when characters are received at communications port *n%*, which can be either 1 or 2.

If your program receives data using an asynchronous communications adapter, the BASIC command-line option /C can be used to set the size of the data buffer.

For an example of the **ON COM** statement, see the **COM** statements programming example.

Using ON KEY

You can use **ON KEY**(*n%*) to specify a routine to branch to when key *n%* is pressed. The argument *n%* is the number of a function key, direction key, or user-defined key.

Keys are processed in the following order:

1. The line printer's echo-toggle key is processed first. Defining this key as a user-defined key trap does not prevent characters from being echoed to the line printer when pressed.
2. Function keys and the direction keys are examined next. Defining a function key or direction key as a user-defined key trap has no effect because these keys are predefined.
3. Finally, the user-defined keys are examined.

The **ON KEY** statement can trap any key, including break or system reset. This makes it possible to prevent accidentally breaking out of a program or rebooting the machine.

Note

After a key is trapped, the information on which key was trapped is no longer available. This means that you cannot subsequently use the **INPUT** statement or **INKEY$** function to find out which key caused the trap. Because you do not know which key press caused the trap, you must set up a different event-handling routine for each key you want to trap.

For an example of the **ON KEY** statement, see the **KEY** statements programming example.

Using ON PEN

You can use **ON PEN** to specify a routine to branch to when the lightpen is activated.

ON PEN is not available for OS/2 protected mode.

For an example of the **ON PEN** statement, see the **PEN** statements programming example.

Using ON PLAY

You can use **ON PLAY**(*queuelimit%*) to specify a routine to branch to when there are fewer than *queuelimit%* notes in the background-music queue. The argument *queuelimit%* is an integer expression between 1 and 32, inclusive.

The following three rules apply to the use of **ON PLAY**:

1. A play-event trap occurs only when music is playing in the background. Play-event traps do not occur when music is running in the foreground.
2. A play-event trap does not occur if the background music queue has already gone from having *queuelimit%* to *queuelimit%*–1 notes when a **PLAY ON** is executed.
3. If *queuelimit%* is a large number, event traps may occur often enough to slow down the program.

ON PLAY is not available for OS/2 protected mode.

For an example of the **ON PLAY** statement, see the **PLAY** function programming example.

Using ON SIGNAL

You can use **ON SIGNAL** to specify a routine to branch to when an OS/2 protected-mode signal is trapped.

The following table lists the numbers of the protected-mode signals:

Number	Signal
1	Ctrl+C
2	Pipe connection broken
3	Program terminated
4	Ctrl+Break
5	Process flag A
6	Process flag B
7	Process flag C

Process flag A, process flag B, and process flag C are used for communicating between processes. Although BASIC has no built-in mechanism for activating a process flag, you can activate the signal handler in a separate process by invoking the OS/2 function DOSFLAGPROCESS.

The **ON SIGNAL** statement is available only for OS/2 protected mode.

For an example of the **ON SIGNAL** statement, see the **SIGNAL** statements programming example.

Using ON STRIG

You can use **ON STRIG**(*n%*) to specify a routine to branch to when the button on the joystick is pressed. The argument *n%* is the trigger number as defined in the following table:

n%	**Button**	**Joystick**
0	Lower	First
2	Lower	Second
4	Upper	First
6	Upper	Second

ON STRIG is not available for OS/2 protected mode.

For an example of the **ON STRIG** statement, see the **STRIG** statements programming example.

Using ON UEVENT

You can use **ON UEVENT** to specify a routine to branch to when a user-defined event occurs. The event usually is a hardware interrupt (although it also can be a software interrupt).

When using DOS, at least two (and sometimes three) pieces of code are needed. The first is the interrupt-service routine. The second is an initialization routine to insert the address of the service routine into the interrupt-vector table. The third is the routine your BASIC program calls to retrieve the data (if any) collected by the interrupt-service routine. For more information, see "Event Handling" in the *Programmer's Guide*.

If the initialization routine "steals" an interrupt used by another service routine, the original address must be restored before your program terminates.

These routines usually are written in assembly language. However, any language whose compiler can generate interrupt-service routines and whose object code can be linked with BASIC can be used.

There are four steps in creating a user-defined event:

1. Write an event-handling routine and add it to your BASIC program.
2. Execute the **ON UEVENT** statement to specify the user-event handling routine.
3. Execute the **UEVENT ON** statement to enable user-event trapping.
4. Call the interrupt-initialization routine to insert the address of the interrupt-service routine into the interrupt-vector table.

You're now ready for the interrupt when it occurs. The interrupt transfers execution to the interrupt-service routine. The service routine collects and stores the data the user wants. It then calls **SetUEvent** (see the entry for **SetUEvent** for details).

SetUEvent sets a flag checked by BASIC before going to the next BASIC statement (or label if executing compiled code using /W instead of /V). When the flag is set, control transfers to the event-handling routine designated in **ON EVENT**.

For an example of the **ON UEVENT** statement, see the **UEVENT** statements programming example.

See Also **COM**, **EVENT**, **KEY** (Event Trapping), **PEN**, **PLAY** (Event Trapping), **RETURN**, **SIGNAL**, **STRIG**, **TIMER**, **UEVENT**

ON...GOSUB, ON...GOTO Statements

Action Branches to one of several specified lines, depending on the value of an expression.

Syntax 1 **ON** *expression* **GOSUB** *line-label-list*

Syntax 2 **ON** *expression* **GOTO** *line-label-list*

Remarks The argument *expression* can be any numeric expression between 0 and 255, inclusive (*expression* is rounded to an integer before the **ON...GOSUB** or **ON...GOTO** statement is evaluated). The argument *line-label-list* consists of a list of line numbers or line labels, separated by commas. The value of *expression* determines which line the program branches to. For example, if the value is 3, the third line specified in the list is the destination of the branch.

The value of *expression* should be greater than or equal to 1 and less than or equal to the number of items in the list. If the value falls outside this range, one of the following results occurs:

Value	Result
Number equal to 0 or greater than number of items in list	Control drops to the next BASIC statement.
Negative number or a number greater than 255	BASIC generates the error message `Illegal function call.`

You may mix line numbers and labels in the same list.

Note You can mix line numbers and labels in the same list. The **ON...GOTO** statement accepts a maximum of 60 line labels or line numbers. The **SELECT CASE** statement provides a more powerful, convenient, and flexible way to perform multiple branching.

See Also **GOSUB, RETURN, SELECT CASE**

Example

The following example causes program control to branch to one of three subroutines, depending on the value of `Chval`:

```
CLS    ' Clear screen.
Attend = 20
Fees = 5 * Attend
PRINT "1  Display attendance at workshops"
PRINT "2  Calculate total registration fees paid"
PRINT "3  End program"
PRINT : PRINT "What is your choice?"
Choice:
    DO
       ch$ = INKEY$
    LOOP WHILE ch$ = ""
    Chval = VAL(ch$)
    IF Chval > 0 AND Chval < 4 THEN
        ON Chval GOSUB Shop, Fees, Progend
    END IF
END
Shop:
    PRINT "ATTENDANCE IS", Attend
    RETURN Choice
Fees:
    PRINT "REGISTRATION FEES ARE $"; Fees
    RETURN Choice
Progend:
    END
```

OPEN Statement (File I/O)

Action Enables I/O to a file, device, or ISAM table.

Syntax 1
OPEN *file$* [[**FOR** *mode*]] [[**ACCESS** *access*]] [[*lock*]] **AS** [[#]]*filenumber%* [[**LEN**=*reclen%*]]
OPEN *database$* **FOR ISAM** *tabletype tablename$* **AS** [[#]]*filenumber%*

Syntax 2
OPEN *mode$*, [[#]]*filenumber%*,*file$*[[, *reclen%*]]

Remarks You must open a file before any I/O operation can be performed on it. **OPEN** allocates a buffer for I/O to the file or device and determines the mode of access used with the buffer.

The **OPEN** statement uses the following arguments:

Argument	Description
file$	A filename or path.
mode	A keyword that specifies one of the following file modes: **RANDOM**, **BINARY**, **INPUT**, **OUTPUT**, or **APPEND**
access	A keyword that specifies the operation performed on the open file: **Read**, **Write**, or **Read Write**
lock	A keyword that specifies the lock type: **Shared**, **Lock Read**, **Lock Write**, or **Lock Read Write**
filenumber%	An integer expression with a value between 1 and 255, inclusive. When an **OPEN** statement is executed, the file number is associated with the file as long as it is open. Other I/O statements may use the number to refer to the file.
reclen%	For random-access files, the record length; for sequential files, the number of characters buffered. The *reclen%* is an integer expression less than or equal to 32,767 bytes.
database$	A database filename or path.
tabletype	The name of a type of table defined using the **TYPE** statement.
tablename$	The name of the ISAM table being opened. It follows the ISAM naming conventions (listed later in this entry).

Syntax 1

The arguments *file$* and *database$* specify an optional device, followed by a filename or path conforming to the DOS file-naming conventions. The argument *database$* must be the name of an ISAM file.

For sequential files, if *file$* does not exist, it is created when opening a file for anything but input—that is, **OUTPUT**, **RANDOM**, **BINARY**, and **APPEND**. For ISAM, a nonexistent *database$* or *tablename$* is created only when using the PROISAMD library; with the PROISAM library, BASIC generates the error message `File Not Found` if *database$* or *tablename$* does not exist.

The argument *mode* is a keyword that specifies one of the following file modes:

Keyword	*Description*
RANDOM	Specifies random-access file mode, the default mode. In random mode, if no **ACCESS** clause is present, three attempts are made to open the file when the **OPEN** statement is executed. Access is attempted in the following order: 1. Read and write 2. Write only 3. Read only
BINARY	Specifies binary file mode. In binary mode, you can read or write information to any byte position in the file using **GET** and **PUT**. In binary mode, if no **ACCESS** clause is present, three attempts are made to open the file, following the same order as for random files.
INPUT	Sequential input mode.
OUTPUT	Sequential output mode.
APPEND	Sequential output mode. Sets the file pointer to the end-of-file and the record number to the last record of the file. A **PRINT #** or **WRITE #** statement then extends (appends to) the file.

If the *mode* argument is omitted, the default random-access mode is assumed.

The argument *access* is a keyword that specifies the operation performed on the opened file. If the file is already opened by another process and the specified type of access is not allowed, the **OPEN** operation fails and BASIC generates the error `Permission denied.` The **ACCESS** clause works in an **OPEN** statement only if you are using a version of DOS that supports networking (DOS version 3.0 or later). In addition, you must run the SHARE.EXE program (or the network startup program must run it) to perform any locking operation. If **ACCESS** is used with **OPEN**, earlier versions of DOS return the error message `Feature unavailable.`

The argument *access* can be one of the following:

Access type	*Description*
Read	Opens the file for reading only.
Write	Opens the file for writing only.
Read Write	Opens the file for both reading and writing. This mode is valid only for random and binary files, and files opened for append.

The *lock* clause works in a multiprocessing environment to restrict access by other processes to an open file. The argument can be one of the following keywords specifying the lock type:

Lock type	Description
Shared	Any process on any machine may read from or write to this file. Do not confuse the shared lock type with the **SHARED** statement or the **SHARED** attribute that appears in other statements.
Lock Read	No other process is granted read access to this file. This access is granted only if no other process has a previous read access to the file.
Lock Write	No other process is granted write access to this file. This lock is granted only if no other process has a previous write access to the file.
Lock Read Write	No other process is granted either read or write access to this file. This access is granted only if read or write access has not already been granted to another process, or if a lock read or lock write is not already in place.

If you do not specify a lock type, the file may be opened for reading and writing any number of times by this statement, but other operations are denied access to the file while it is opened.

When **OPEN** is restricted by a previous process, BASIC generates error 70, `Permission denied`.

The argument *reclen%* is an integer expression less than or equal to 32,767 bytes. It specifies different settings for random-access or sequential files:

For random-access:	*For sequential files:*
The argument *reclen%* sets the record length (the number of characters in a record).	The argument *reclen%* specifies the number of characters to be loaded into the buffer before the buffer is written to, or read from, the disk.
The default is 128 bytes.	A larger buffer means more room taken from BASIC, but faster file I/O. A smaller buffer means more room in memory for BASIC, but slower I/O.
	The default is 512 bytes.

The **LEN** clause and *reclen%* are ignored if *mode* is **BINARY**. For sequential files, *reclen%* need not correspond to an individual record size, because a sequential file may have records of different sizes.

ISAM is a keyword that specifies you are opening an ISAM table.

The argument *tabletype* is the name of a user-defined type that defines a subset of the table definition. (See the entry for the **TYPE** statement.) The argument *tablename$* is the name of the ISAM table being opened. It follows the ISAM naming conventions:

- It has no more than 30 characters.
- It uses only alphanumeric characters (A-Z, a-z, 0-9).
- It begins with an alphabetic character.
- It includes no BASIC special characters.

Syntax 2

In the second (alternative) form of the **OPEN** syntax, *mode$* is a string expression that begins with one of the following characters and specifies the file mode:

Mode	Description
O	Sequential output mode.
I	Sequential input mode.
R	Random-access file I/O mode.
B	Binary file mode.
A	Sequential output mode. Sets the file pointer to the end of the file and the record number to the last record of the file. A **PRINT #** or **WRITE #** statement extends (appends to) the file.

Note This alternative syntax does not support any of the access and file-sharing options found in the primary syntax. It is supported for compatibility with programs written in earlier versions of BASIC.

The following devices are supported by BASIC and can be named and opened with the argument *file$*:

KYBD, SCRN, COM*n*, LPT*n*, CONS, PIPE.

The BASIC file I/O system allows you to take advantage of user-installed devices. (See your DOS manual for information on character devices.)

Character devices are opened and used in the same manner as disk files. However, characters are not buffered by BASIC as they are for disk files. The record length for device files is set to 1.

BASIC sends only a carriage return at the end of a line. If the device requires a line feed, the driver must provide it. When writing to device drivers, keep in mind that other BASIC users will want to read and write control information. Writing and reading of device-control data is handled by the **IOCTL** statement and **IOCTL$** function.

None of the BASIC devices directly supports binary mode. However, line-printer devices can be opened in binary mode by adding the **BIN** keyword:

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1
```

Opening a printer in binary mode eliminates printing a carriage return at the end of a line.

When you open an ISAM table, the next record is the first record in the table and the current index is the NULL index.

Any ISAM operation that closes a table causes transactions to be committed. For example, if a type mismatch occurs while you are opening an ISAM table, the table is closed and a pending transaction is committed.

You may wish to code your programs so they first open all tables, then perform all transactions, then close tables. Make sure any operation that can close a table occurs outside a transaction.

Note In input, random-access, and binary modes you can open a file under a different file number without first closing the file. In output or append mode you must close a file before opening it with a different file number.

See Also **CLOSE, FREEFILE, TYPE**

Example See the **FREEFILE** function programming example, which uses the **OPEN** statement.

OPEN COM Statement

Action Opens and initializes a communications channel for I/O.

Syntax **OPEN "COM***n*: *optlist1 optlist2*" ⟦**FOR** *mode*⟧ **AS** ⟦#⟧*filenum%* ⟦**LEN=***reclen%*⟧

Remarks The **OPEN COM** statement must be executed before a device can be used for communication using an RS232 interface. **COM***n* is the name of the device to be opened. If there are any syntax errors in the **OPEN COM** statement, BASIC generates the error message `Bad file name.`

The **OPEN COM** statement uses the following arguments:

Argument	Description
n	The communications port to open, such as 1 for COM1 and 2 for COM2.
optlist1	The most-often-used communications parameters. The defaults are 300 baud, even parity, 7 data bits, and 1 stop bit. The syntax of *optlist1* and options are described later in this entry.
optlist2	Up to 10 other optional, less-often-used data communications parameters. The parameters for *optlist2* are described later in this entry.
mode	One of the keywords **OUTPUT**, **INPUT**, or **RANDOM** (the default).
filenum%	Any unused file number between 1 and 255, inclusive.
reclen%	The size of a random-access-mode buffer (128 bytes is the default).

The argument *optlist1* specifies the communication-line parameters and has this syntax:

⟦*speed*⟧ ⟦,⟦*parity*⟧ ⟦,⟦*data*⟧ ⟦,⟦*stop*⟧⟧⟧⟧

The following table lists the valid values for *optlist1* options:

Option	**Description**	**Range**	**Default**
speed	Baud rate (bits per second) of the device to be opened	75, 110, 150, 300, 600, 1200, 1800, 2400, 9600	300
parity	Method of parity checking (none, even, odd, space, mark)	N, E, O, S, M	E
data	Number of data bits per byte	5, 6, 7, or 8	7
stop	Number of stop bits	1, 1.5, or 2	1[1]

[1] The default value is 1 for baud rates greater than 110. For baud rates less than or equal to 110, the default value is 1.5 when *data* is 5; otherwise, the value is 2.

Options in this list must be entered in the order shown. Use comma placeholders for defaults. For example:

```
OPEN "COM1: ,N,8," FOR INPUT AS #1
```

Only the baud rates shown are supported. Any other value for *speed* is invalid.

If any options from *optlist2* are chosen, comma placeholders still must be used even if all of the *optlist1* options are defaults. For example:

```
OPEN "COM1: ,,,,CD1500" FOR INPUT AS #1
```

If you set *data* to 8 bits per byte, you must specify no parity (N).

Note Because BASIC uses complete bytes (8 bits) for numbers, you must specify 8 data bits when transmitting or receiving numeric data.

The *optlist2* options can be specified in any order, and must be separated by commas. There are three types of options: data mode, buffer size, and handshaking.

The data-mode options (**ASC**, **BIN**, and **LF**) are described in the following table:

Option	Description
ASC	Opens the device in ASCII mode. In ASCII mode, tabs are expanded to blanks, carriage returns are forced at end-of-line, and Ctrl+Z means end-of-file. When the channel is closed, Ctrl+Z is sent over the RS-232 line.
BIN	Opens the device in binary mode. This option supersedes the LF option. BIN is elected by default unless ASC is specified. In binary mode, tabs are not expanded to spaces, a carriage return is not forced at the end of a line, and Ctrl+Z is not treated as end-of-file. When the channel is closed, Ctrl+Z will not be sent over the RS232 line.
LF	Allows communication files to be printed on a serial line printer. Effective only with the ASC option. When LF is specified, a line-feed character (0AH) is automatically sent after each carriage-return character (0DH). This includes the carriage return sent as a result of the width setting. Note that **INPUT** and **LINE INPUT**, when used to read from a communications file that was opened with the LF option, stop when they see a carriage return, ignoring the line feed.

The buffer-size options for sequential modes (RB and TB) are described in the following table:

Option	Description
RB⟦*n*⟧	Sets the size of the receive buffer to *n* bytes. The initial buffer size, if *n* or the RB option is omitted, is 512 bytes, unless overridden by the /C option on the QBX or BC command line. The default value, if *n* or the RB option is omitted, is the current receive-buffer size. The maximum size is 32,767 bytes.
TB⟦*n*⟧	Sets the size of the receive buffer to *n* bytes. The initial buffer size, if *n* or the TB option is omitted, is 512 bytes. The default value, if *n* or the TB option is omitted, is the current receive-buffer size.

Handshake and timing options (RS, CD, CS, DS, and OP) are described in the table below:

Option	Description
RS	Suppresses detection of Request To Send (RTS).
CD⟦*m*⟧	Specifies the timeout period on the Data Carrier Detect line (DCD). If no signal appears on the DCD line (the DCD line remains low) for more than *m* milliseconds, a device timeout occurs. If a CD timeout occurs, **ERDEV** contains 130 (82H). The range for *m*, if specified, is 0–65,535 milliseconds, inclusive. The default value is 0, if the CD option is not used, or if *m* is omitted. When *m* is 0, either by default or assignment, it means that the state of the DCD line is to be ignored.
CS⟦*m*⟧	Specifies the timeout period on the Clear To Send line (CTS). If no signal appears on the CTS line (the CTS line remains low) for more than *m* milliseconds, a device timeout occurs. If a CS timeout occurs, **ERDEV** contains 128 (80H). The range for *m*, if specified, is 0–65,535 milliseconds, inclusive. The default value is 1,000 milliseconds, if the CS option is not used, or if *m* is omitted. When *m* is 0, either by default or assignment, it means the state of the CTS line is to be ignored.
DS⟦*m*⟧	Specifies the timeout period on the Data Set Ready line (DSR). If no signal appears on the DSR line (the DSR line remains low) for more than *m* milliseconds, a device timeout occurs. If a DS timeout occurs, **ERDEV** contains 129 (81H). The range for *m*, if specified, is 0–65,535 milliseconds, inclusive. The default value is 1,000 milliseconds, if the **DS** option is not used, or if *m* is omitted. When *m* is 0, either by default or assignment, it means that the state of the DSR line is to be ignored.
OP⟦*m*⟧	Specifies how long the **OPEN** statement waits for all communications lines to become active. The range for *m* is 0–65,535 milliseconds, inclusive. The default value for *m*, if the OP option is not used, is 10 times the CD or DS timeout value, whichever is greater. If OP is specified, but with *m* omitted, **OPEN COM** waits for 10 seconds. Use a relatively large value for the OP option compared to the CS, DS, or CD options.

The argument *filenum%* is the number used to open the file.

The argument *reclen%* is effective only in random-access mode and specifies the length of the random-access buffer (default is 128 bytes). You can use any of the random-access I/O statements, such as **GET** and **PUT**, to treat the device as if it were a random-access file.

The argument *mode* is one of the following string expressions:

Mode	Description
OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
RANDOM	Specifies random-access mode.

If *mode* is omitted, it is assumed to be random-access input/output.

The **OPEN COM** statement performs the following steps in opening a communications device:

1. The communications buffers are allocated and interrupts are enabled.
2. The Data Terminal Ready line (DTR) is set high.
3. If either of the OP or DS options is nonzero, the statement waits for the timeout period for the Data Set Ready line (DSR) to go high. If a timeout occurs, the process goes to step 6.
4. If the RS option is not specified, the Request To Send line (RTS) is set high.
5. If either of the OP or CD options is nonzero, the statement waits for the timeout period for the Data Carrier Detect line (DCD) to go high. If a timeout occurs, the process goes to step 6. Otherwise, the RS232 device has been successfully opened.
6. If there is a timeout, the open fails. The process deallocates the buffers, disables interrupts, clears all of the control lines, and generates the message `Device timeout.` In addition, for DOS, the process sets the value of **ERDEV$** to COM and sets **ERDEV** to a value that indicates the signal line that timed out, according to the following table:

ERDEV value	Signal line
128 (80H)	Clear to Send (CTS) timeout
129 (81H)	Data Set Ready (DSR) timeout
130 (82H)	Data Carrier Detect (DCD) timeout

Note If there is not an **OPEN COM** statement in your program, you can reduce the size of the .EXE file by linking your program with the stub file NOCOM.OBJ.

Example See the **COM** statements programming example, which uses the **OPEN COM** statement.

OPTION BASE Statement

Action Declares the default lower bound for array subscripts.

Syntax **OPTION BASE** *n%*

Remarks The **OPTION BASE** statement is never required.

The value of *n%* must be either 0 or 1. The default base is 0. If the following statement is executed, the default lowest value of an array subscript is 1.

```
OPTION BASE 1
```

Note The **TO** clause in the **DIM** statement provides an easier, more flexible way to control the range of an array's subscripts. If the lower bound of an array subscript is not explicitly set, then **OPTION BASE** can be used to change the default lower bound to 1.

The **OPTION BASE** statement can be used only once in a module (source file) and can appear only in the module-level code. An **OPTION BASE** statement must be used before you can declare the dimensions for any arrays.

Chained programs can have an **OPTION BASE** statement if no arrays are passed between them in a **COMMON** block, or if the specified base is identical in the chained programs. The chained-to program inherits the **OPTION BASE** value of the chaining program if **OPTION BASE** is omitted in the latter.

See Also **DIM, LBOUND, UBOUND**

Example The following example shows the use of **OPTION BASE** to override the default base array subscript value of 0. Subscripts in array `A` range from 1 to 20 rather than 0 to 19.

```
OPTION BASE 1
DIM A(20)
PRINT "The base subscript in array A is"; LBOUND(A)
PRINT "The upper bound subscript in array A is"; UBOUND(A)
```

Output

```
The base subscript in array A is 1
The upper bound subscript in array A is 20
```

OUT Statement

Action Sends a byte to a hardware I/O port.

Syntax **OUT** *port*, *data%*

Remarks The **OUT** statement complements the **INP** function, which returns the byte read from a hardware I/O port. The **OUT** statement uses the following arguments:

Argument	Description
port	A numeric expression with an integer value between 0 and 65,535, inclusive, that identifies the destination hardware I/O port.
data%	A numeric expression with an integer value between 0 and 255, inclusive, that is the data to be sent to the port.

The **OUT** statement and the **INP** function give a BASIC program direct control over the hardware in a system through the I/O ports. **OUT** and **INP** must be used carefully because they directly manipulate the system hardware.

Note The **OUT** statement is not available in OS/2 protected mode.

See Also **INP**, **WAIT**

Example The following example uses the **OUT** and **INP** statements to control the timer and speaker to produce a note:

```
' Play a scale using speaker and timer.
CONST WHOLE = 5000!, QRTR = WHOLE / 4!
CONST C = 523!, D = 587.33, E = 659.26, F = 698.46, G = 783.99
CONST A = 880!, B = 987.77, C1 = 1046.5
CALL Sounds(C, QRTR): CALL Sounds(D, QRTR)
CALL Sounds(E, QRTR): CALL Sounds(F, QRTR)
CALL Sounds(G, QRTR): CALL Sounds(A, QRTR)
CALL Sounds(B, QRTR): CALL Sounds(C1, WHOLE)

SUB Sounds (Freq!, Length!) STATIC
' Ports 66, 67, and 97 control timer and speaker.
    ' Divide clock frequency by sound frequency
    ' to get number of "clicks" clock must produce.
    Clicks% = CINT(1193280! / Freq!)
    LoByte% = Clicks% AND &HFF
    HiByte% = Clicks% \ 256
    OUT 67, 182                     ' Tell timer that data is coming.
    OUT 66, LoByte%                 ' Send count to timer.
    OUT 66, HiByte%
    SpkrOn% = INP(97) OR &H3        ' Turn speaker on by setting
    OUT 97, SpkrOn%                 ' bits 0 and 1 of PPI chip.
    FOR I! = 1 TO Length!: NEXT I!  ' Leave speaker on.
    SpkrOff% = INP(97) AND &HFC     ' Turn speaker off.
    OUT 97, SpkrOff%
END SUB
```

PAINT Statement

Action Fills a graphics area with the color or pattern specified.

Syntax **PAINT** ⟦**STEP**⟧ (*x!*,*y!*) ⟦,⟦*paint*⟧ ⟦,⟦*bordercolor&*⟧ ⟦,*background$*⟧⟧⟧

Remarks The following list describes the parts of the **PAINT** statement:

Part	Description
STEP	Keyword indicating that coordinates are relative to the most recently plotted point. For example, if the last point plotted were (10,10), then the coordinates referred to by `STEP (4,5)` would be (4+10, 5+10) or (14,15).
(*x!*,*y!*)	The coordinates where painting begins. The point must be inside or outside of a figure, not on the border itself. If this point is inside, the figure's interior is painted; if the point is on the outside, the background is painted.
paint	A numeric or string expression. If *paint* is a numeric expression, then the number must be a valid color attribute. The corresponding color is used to paint the area. If you do not specify *paint,* the foreground color attribute is used. (See the **COLOR**, **PALETTE**, and **SCREEN** statements for discussions of valid colors, numbers, and attributes.) If the argument *paint* is a string expression, **PAINT** "tiles," a process that paints a pattern rather than a solid color. Tiling is similar to "line styling," which creates dashed lines rather than solid lines.
bordercolor&	A numeric expression that identifies the color attribute to use to paint the border of the figure. When the border color is encountered, painting of the current line stops. If the border color is not specified, the *paint* argument is used.
background$	A string value that gives the "background tile slice" to skip when checking for termination of the boundary. Painting is terminated when adjacent points display the paint color. Specifying a background tile slice allows you to paint over an already painted area. When you omit *background$* the default is `CHR$ (0)`.

Painting is complete when a line is painted without changing the color of any pixel; in other words, when the entire line is equal to the paint color. The **PAINT** statement permits coordinates outside the screen or viewport.

"Tiling" is the design of a **PAINT** pattern represented in string form. The tile string is eight bits wide and up to 64 bytes long. Each byte masks eight bits along the x axis when plotting points. The syntax for constructing the tile mask is

A$ = **CHR$**(*arg1*)+**CHR$**(*arg2*)+...+**CHR$**(*argn*)
PAINT (*x*,*y*), A$

The arguments to **CHR$** are numbers between 0 and 255, represented in binary form across the x axis of the tile. There can be up to 64 of these **CHR$** elements; each generates an image not of the assigned character, but of the bit arrangement of the code for that character. For example, the decimal number 85 is binary 01010101; the graphic image line on a black-and-white screen generated by `CHR$(85)` is an eight-pixel line, with even-numbered points white and odd-numbered points black. That is, each bit equal to 1 turns the associated pixel on and each bit equal to 0 turns the associated bit off in a black-and-white system. The ASCII character `CHR$(85)`, which is U, is not displayed in this case.

When supplied, *background$* defines the "background tile slice" to skip when checking for boundary termination. You cannot specify more than two consecutive bytes that match the tile string in the background tile slice. If you specify more than two consecutive bytes, BASIC generates an error message `Illegal function call.`

Tiling also can be done to produce various patterns of different colors. See Chapter 5, "Graphics" in the *Programmer's Guide* for a complete description of how to do tiling.

See Also **CHR$**, **CIRCLE**, **DRAW**, **LINE**, **SCREEN** Statement

Example The following example uses **PAINT** to create a magenta fish with a cyan tail:

```
CONST PI = 3.1415926536
CLS       ' Clear screen.
SCREEN 1

CIRCLE (190, 100), 100, 1, , , .3    ' Outline fish body in cyan.
CIRCLE (265, 92), 5, 1, , , .7       ' Outline fish eye in cyan.
PAINT (190, 100), 2, 1               ' Fill in fish body with magenta.

LINE (40, 120)-STEP (0, -40), 2      ' Outline tail in magenta.
LINE -STEP (60, 20), 2
LINE -STEP (-60, 20), 2
PAINT (50, 100), 1, 2                ' Paint tail cyan.

CIRCLE (250,100),30,0,PI*3/4,PI* 5/4,1.5   ' Draw gills in black.
FOR Y = 90 TO 110 STEP 4
   LINE (40, Y)-(52, Y), 0                 ' Draw comb in tail.
NEXT
```

PALETTE, PALETTE USING Statements

Action Change one or more colors in the palette.

Syntax **PALETTE** ⟦*attribute%*, *color&*⟧
PALETTE USING *array-name* ⟦(*array-index*)⟧

Remarks The **PALETTE** and **PALETTE USING** statements use the following arguments:

Argument	Description
attribute%	The palette attribute to be changed.
color&	The display color number to be assigned to the attribute. The *color&* value must be a long-integer expression for the IBM Video Graphics Array adapter (VGA) and IBM Multicolor Graphics Array adapter (MCGA) in screen modes 11 to 13. Integer or long-integer expressions can be used with the IBM Enhanced Graphics Adapter (EGA).
array-name	An array that contains more than one display color. It must be a long-integer array for VGA and MCGA adapters in screen modes 11 to 13. Otherwise, it can be either an integer or long-integer array.
array-index	The index of the first array element to use in setting the palette. The arguments *array-name* and *array-index* are used to change more than one palette assignment with a single **PALETTE USING** statement.

The **PALETTE** statement works *only* on systems equipped with the EGA, VGA, or MCGA adapters. The **PALETTE** statement is not supported in screen modes 3 or 4.

The **PALETTE** statement provides a way of mapping display colors (the actual binary values used by the adapter) to color attributes (a smaller set of values).

When a program enters a screen mode, the attributes are set to a series of default color values. (See the **SCREEN** statement for a list of the default colors.) In the EGA, VGA, and MCGA adapters, the default values have been selected so the display shows the same colors, even though the EGA uses different color values.

With the **PALETTE** statement you can assign different colors to the attributes. Changing the display color assigned to an attribute immediately changes those colors currently displayed on the screen associated with that attribute.

For example, assume that the current palette contains colors 0, 1, 2, and 3 in the four attributes numbered 0, 1, 2, and 3. The following **DRAW** statement selects attribute 3, and draws a line of 100 pixels using the display color associated with attribute 3, in this case also 3:

```
DRAW "C3L100"
```

If the following statement is executed, the color associated with attribute 3 is changed to color 2:

```
PALETTE 3,2
```

All text or graphics currently on the screen displayed using attribute 3, including the line that is 100 pixels long, are instantaneously changed to color 2. Text or graphics subsequently displayed with attribute 3 also are displayed in color 2. The new palette of colors contains 0, 1, 2, and 2. A **PALETTE** statement with no arguments sets the palette back to the default color values.

With **PALETTE USING**, all entries in the palette can be modified in one statement. Each attribute in the palette is assigned a corresponding color from the array.

The dimensions for the array must be large enough to set all the palette entries after *array-index*. For example, if you are assigning colors to a palette with 16 attributes and the *array-index* argument is 5 (the first array element to use in resetting the palette), then the dimensions for the array must be declared to hold at least 20 elements (because the number of elements from 5 to 20, inclusive, is 16):

```
DIM PAL%(20)
.
.
.
PALETTE USING PAL%(5)
```

Note that a *color&* argument of –1 in the array leaves the attribute unchanged in the palette. All other negative numbers are invalid values for color.

Attribute 0 is always the screen background color. Under a common initial palette setting, points colored with the attribute 0 appear black on the display screen. Using the **PALETTE** statement, you could, for example, change the mapping of attribute 0 from black to white. You can also use the **COLOR** statement to change the screen background color in modes 1 and 7 through 10.

The 64 EGA colors are derived from four levels each of red, green, and blue. For example, black is composed of red, green, and blue levels of (0,0,0), bright white is (3,3,3), dark gray is (1,1,1), and so on. The best way to see the Microsoft BASIC color code (0–63) associated with each combination of red, green, and blue levels is to run the following program:

```
' Display the EGA color codes 1 through 63
' using color code 0 (black) as background.
DEFINT A-Z
SCREEN 9                          ' Establish EGA screen mode.
' Display a set of nine color bars.
FOR ColorCode% = 1 TO 9
   LINE (((ColorCode% * 64) - 24), 40)-STEP(60, 140), ColorCode%, BF
NEXT ColorCode%
' Display seven sets of nine color bars.
' A new set is displayed each time user presses a key.
FOR Set% = 0 TO 6
   FOR ColorBar% = 1 TO 9
       DisplayCode% = (Set% * 9) + ColorBar%
       LOCATE 15, (ColorBar% * 8)
          PRINT DisplayCode%
       PALETTE ColorBar%, DisplayCode%
   NEXT ColorBar%
   SLEEP
NEXT Set%
END
```

The following table lists attribute and color ranges for various adapter types and screen modes. See the **SCREEN** statement for the list of colors available for various screen modes, monitor, and graphics-adapter combinations.

Table 1.9 Attribute and Color Ranges for Adapter Types and Screen Modes

Screen mode	Monitor attached	Adapter	Attribute range	Color range	PALETTE supported
0	Monochrome	MDPA	0–15		No
	Monochrome	EGA	0–15	0–2	Yes
	Color	CGA	0–15		No
	Color/Enhanced[1]	EGA	0–15	0–15 / 0–63	Yes
	Analog	VGA	0–15	0–63	Yes
	Color/Analog	MCGA	0–15	0–63	No
1	Color	CGA	0–3	0–15[2]	No
	Color/Enhanced[1]	EGA	0–3	0–15	Yes
	Analog	VGA	0–3	0–15	Yes
	Color/Analog	MCGA	0–3	0–15	No
2	Color	CGA	0–1		No
	Color/Enhanced[1]	EGA	0–1	0–15	Yes
	Analog	VGA	0–1	0–15	Yes
	Color/Analog	MCGA	0–1	0–15	No
3	Monochrome	HGC	0–1		No
4	Color/Enhanced	OCGA/OEGA/ OVGA	0–1	0–15[3]	No
7	Color/Enhanced[1]	EGA	0–15	0–15	Yes
		VGA	0–15	0–15	Yes
8	Color/Enhanced[1]	EGA	0–15	0–15	Yes
		VGA	0–15	0–15	Yes
9	Enhanced[1]	EGA[4]	0–3	0–63	Yes
	Enhanced[1]	EGA[5]	0–15	0–63	Yes
	Analog	VGA	0–16	0–63	Yes
10	Monochrome	EGA	0–3	0–8	Yes
	Analog	VGA	0–3	0–8	Yes
11	Analog	VGA	0–1	0–262,143[6]	Yes
	Analog	MCGA	0–1	0–262,143[6]	Yes
12	Analog	VGA	0–15	0–262,143[6]	Yes
13	Analog	VGA	0–255	0–262,143[6]	Yes
	Analog	MCGA	0–255	0–262,143[6]	Yes

1 IBM Enhanced Color Display.
2 Color range available for attribute 0 only.
3 Color range available for attribute 1 only.
4 With 64K of EGA memory.
5 With more than 64K of EGA memory.
6 Range of display colors is actually from 0 to 4,144,959, but only 262,144 of these can be displayed.

To calculate a VGA color value, select the intensities of red, green, and blue. The intensity of a color is a number from 0 (low intensity) to 63 (high intensity). Then use the following formula to calculate the actual color number:

color number = 65,536 * blue + 256 * green + red

This formula yields integer values from 0 to 4,144,959, but because there are gaps in the range of color numbers, you should use the formula rather than just select a number.

When used with the IBM Analog Monochrome Monitor, the VGA color values are converted to a gray-scale value by taking a weighted sum of the red, blue, and green intensities:

gray value = 11% blue + 59% green + 30% red

For example, if the blue, green, and red intensities are 45, 20, and 20, the gray intensity value calculation would be gray value = (.11 * 45) + (.59 * 20) + (.30 * 20) = 22. The fractional part of the result is dropped.

See Also **COLOR**, **SCREEN** Statement

Example The following example illustrates the use of the **PALETTE** and **PALETTE USING** statements:

```
DEFINT A-Z
DIM SHARED PaletteArray(15)
CONST ASPECT = 1 / 3
SCREEN 8      ' 640 x 200, 16 color resolution.
' Initialize PaletteArray.
FOR I = 0 TO 15
    PaletteArray(I) = I
NEXT I
' Draw and paint concentric ellipses.
FOR ColorVal = 15 TO 1 STEP -1
    Radius = 20 * ColorVal
    CIRCLE (320, 100), Radius, ColorVal, , , ASPECT
    PAINT (320, 100), ColorVal
NEXT
' Shift the palette until a key is pressed.
DO
    FOR I = 1 TO 15
        PaletteArray(I) = (PaletteArray(I) MOD 15) + 1
    NEXT I
    PALETTE USING PaletteArray(0)
    ' Map the black background to a random color.
    PALETTE 0, PaletteArray(INT(RND * 15))
LOOP WHILE INKEY$ = ""
```

PCOPY Statement

Action Copies one video memory page to another.

Syntax **PCOPY** *sourcepage%*,*destinationpage%*

Remarks The **PCOPY** statement uses the following arguments:

Argument	Description
sourcepage%	A numeric expression with an integer value between 0 and *n* that identifies a video memory page to be copied.
destinationpage%	A numeric expression with an integer value between 0 and *n* that identifies the video memory page to be copied to.

The value of *n* is determined by the current size of video memory and the current screen mode. The number of video memory pages available depends on the current screen mode, the graphics adapter, and how much screen memory is available with the adapter.

Note Multiple video pages are not available in OS/2 protected mode, so the **PCOPY** statement has no effect.

See the **SCREEN** statement for more information about the number of pages available in different modes

See Also **CLEAR**, **SCREEN** Statement

Example See the **SCREEN** statement programming example, which uses the **PCOPY** statement.

PEEK Function

Action Returns byte value stored at a specified memory location. Complements the **POKE** statement.

Syntax **PEEK**(*address*)

Remarks The returned value is an integer between 0 and 255, inclusive. The argument *address* is a value between 0 and 65,535, inclusive. The argument *address* is treated as the offset from the current default segment (as set by the **DEF SEG** statement).

If *address* is a single- or double-precision floating-point value, or a long integer, it is converted to a 2-byte integer.

When using **PEEK** to return a byte from a far-string array, use the **SSEG** and **SADD** functions to obtain the current segment and offset. For example:

```
DEF SEG = SSEG(a$)        ' Set current segment address to address of a$.
StringOffset = SADD(a$)   ' Determine string's location within segment.
PEEK(StringOffset)        ' Return the byte stored at this location.
```

Direct string manipulation with **PEEK** should be used cautiously, because BASIC moves string locations during run time.

PEEK and Expanded Memory Arrays

Do not use **PEEK** to return a byte from an expanded memory array. If you start QBX with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays less than 16K in size.
- Fixed-length string arrays less than16K in size.
- User-defined-type arrays less than 16K in size.

If you want to use **PEEK** to return a byte from an array, first start QBX without the /Ea switch. (Without the /Ea switch, no arrays are stored in expanded memory.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started.*

Note When programming with OS/2 protected mode, note that any address referred to by **PEEK** must be readable. If **PEEK** refers to an address for which your process does not have read permission, the operating system may generate a protection exception, or BASIC may generate the error message `Permission denied`.

See Also **DEF SEG; POKE; SADD; SSEG; SSEGADD; VARPTR, VARSEG; VARPTR$**

Example See the **DEF SEG** statement programming example, which uses the **PEEK** statement.

PEN Function

Action Returns lightpen status.

Syntax PEN(*n%*)

Remarks The argument *n%* is an integer value between 0 and 9, inclusive, that specifies what information is to be returned about the status of the lightpen.

Note The **PEN** function does not work when the mouse driver is enabled because the mouse driver uses the **PEN** function's BIOS calls. Use mouse function 14 to disable the driver's lightpen emulation. Mouse function 13 turns emulation back on. See your mouse manual for more information.

The following list describes the values for *n%* and the corresponding values returned by **PEN**:

Argument	Value returned
0	Whether the pen was down since the last function call (–1 if yes, 0 if no).
1	The x coordinate of the last pen press.
2	The y coordinate of the last pen press.
3	The current pen switch status (–1 if down, 0 if up).
4	The x coordinate where the pen last left the screen.
5	The y coordinate where the pen last left the screen.
6	The character row of the last pen press.
7	The character column of the last pen press.
8	The character row where the pen last left the screen.
9	The character column where the pen last left the screen.

The light-pen coordinate system is identical to the current graphics screen mode, without viewport or window considerations.

Note The **PEN** function is not available in OS/2 protected mode.

See Also **PEN** Statements

Example See the **ON PEN** statement programming example, which uses the **PEN** function.

PEN Statements

Action Enable, disable, or suspend lightpen-event trapping.

Syntax

PEN ON
PEN OFF
PEN STOP

Remarks The **PEN ON** statement enables lightpen-event trapping. A lightpen event occurs whenever the lightpen is activated by pressing the tip to the screen or pressing the touch ring. If a lightpen event occurs after a **PEN ON** statement, the routine specified in the **ON PEN** statement is executed.

The **PEN OFF** statement disables lightpen-event trapping. No trapping takes place until a **PEN ON** statement is executed. Events occurring while trapping is off are ignored.

The **PEN STOP** statement suspends lightpen-event trapping. No trapping takes place until a **PEN ON** statement is executed. Events occurring while trapping is suspended are remembered and processed when the next **PEN ON** statement is executed. However, remembered events are lost if **PEN OFF** is executed.

When a lightpen-event trap occurs (that is, the **GOSUB** is performed), an automatic **PEN STOP** is executed so that recursive traps cannot take place. The **RETURN** statement from the trapping routine automatically performs a **PEN ON** statement unless an explicit **PEN OFF** was performed inside the routine.

A **PEN ON** statement must be executed before you use the **PEN** function. If a **PEN** function is executed when the lightpen is off, BASIC generates the error message `Illegal function call.`

Note The lightpen requires an IBM Color Graphics Adapter.

The **PEN** statements are not available for OS/2 protected mode.

For more information, see Chapter 9, "Event Handling" in the *Programmer's Guide*.

See Also ON *event*

Example

The following example uses the **PEN** statements and the **PEN** function to enable and display current lightpen status (up or down) and position (x and y coordinates). The **ON PEN** statement passes control to the `PenReport` routine when a lightpen event occurs.

```
' Note: Do not run this program with your mouse driver enabled.
CLS                   ' Clear screen.
COLOR 0, 7            ' Set black on white.
CLS                   ' Clear screen.
PEN ON                ' Enable lightpen
ON PEN GOSUB PenReport
DO
    LOCATE 23, 12
    PRINT "Press the lightpen against the screen to see a report."
    LOCATE 24, 17
    PRINT "  Press the Spacebar to exit the program.     ";
LOOP UNTIL INKEY$ = " "
PEN OFF               ' Disable lightpen.
COLOR 7, 0            ' Set back to black on white.
CLS                   ' Clean up the screen.
END

PenReport:
    DO
        P = PEN(3)
        ' Report lightpen status and get X and Y position.
        LOCATE 10, 27
        PRINT "A Pen event has occurred."
        LOCATE 24, 17
        PRINT "Press ANY key to exit the lightpen report.";
        LOCATE 12, 30
        PRINT "lightpen is ";
        IF P THEN
            PRINT "Down"
            X = PEN(4): Y = PEN(5)
        ELSE
            PRINT "Up   "
            X = 0: Y = 0
        END IF
        ' Report the X and Y position.
        LOCATE 14, 22
        PRINT "X Position ="; X; "    "
        LOCATE 14, 40
        PRINT "Y Position ="; Y; "    "
    LOOP WHILE INKEY$ = ""
    RETURN
```

PLAY Function

Action Returns the number of notes currently in the background-music queue.

Syntax **PLAY** (*n*)

Remarks The argument *n* is a dummy argument and can be any numeric value.

PLAY(*n*) will return 0 when music is running in the foreground.

The **PLAY** function is not available for OS/2 protected mode.

See Also **ON** *event*, **PLAY** Statements (Event Trapping), **PLAY** Statement (Music)

Example The following example plays continuous music by calling an event-handling routine when the background music buffer goes from three to two notes:

```
CLS
' Call routine Replay when the music buffer goes
' from 3 to 2 notes.
ON PLAY(3) GOSUB Replay
' Turn on event trapping for PLAY.
PLAY ON
' Define a string containing the melody.
FElise$ = "o3 L8 E D+ E D+ E o2 B o3 D C L2 o2 A"
PLAY "MB X" + VARPTR$(FElise$)
' Suspend event trapping until next PLAY ON but remember
' events that occur while event trapping is disabled.
PLAY STOP
' Introduce a variable-length delay.
LOCATE 23, 1: PRINT "Press any key to continue."
DO
LOOP WHILE INKEY$ = ""
```

```
' Re-enable play-event trapping, remembering that a play event
' has occurred.
PLAY ON
LOCATE 23, 1: PRINT "Press any key to stop.    "
' Loop until a key is pressed.
DO
    GOSUB BackGround
LOOP WHILE INKEY$ = ""
' Disable play-event processing.
PLAY OFF
' Count down to 0 notes in the queue.
DO
    GOSUB BackGround
LOOP UNTIL NoteCount = 0
END

' Play event-handling routine.
Replay:
    ' Increment and print a counter each time.
    Count% = Count% + 1
    LOCATE 3, 1: PRINT "Replay routine called"; Count%; "time(s)";
    ' Play it again to fill the buffer.
    PLAY "MB X" + VARPTR$(FElise$)
RETURN

' Background music queue reporting routine.
BackGround:
    ' Get a note count.
    NoteCount = PLAY(0)
    LOCATE 1, 1
    PRINT "Background queue notes remaining --> "; NoteCount
    ' Loop until Notecount changes or equals 0.
    DO
    LOOP UNTIL NoteCount <> PLAY(0) OR NoteCount = 0
RETURN
```

PLAY Statements (Event Trapping)

Action Enable, disable, and suspend play-event trapping.

Syntax **PLAY ON**
PLAY OFF
PLAY STOP

Remarks The **PLAY ON** statement enables play-event trapping. (A play event occurs when the number of notes in the background music queue drops below the limit you set with the **ON PLAY** statement.) If a play event occurs after a **PLAY ON** statement, the routine specified in the **ON PLAY** statement is executed.

The **PLAY OFF** statement disables play-event trapping. No trapping takes place until a **PLAY ON** statement is executed. Events occurring while trapping is off are ignored.

PLAY STOP suspends play-event trapping. No trapping takes place until a **PLAY ON** statement is executed. Events occurring while trapping is suspended are remembered and processed when the next **PLAY ON** statement is executed. However, remembered events are lost if **PLAY OFF** is executed.

When a play-event trap occurs (that is, the **GOSUB** is performed), an automatic **PLAY STOP** is executed so that recursive traps cannot take place. The **RETURN** operation from the trapping routine automatically executes a **PLAY ON** statement unless an explicit **PLAY OFF** was performed inside the routine.

For more information, see Chapter 9, "Event Handling" in the *Programmer's Guide*.

Note The **PLAY** statements (event trapping) are not available for OS/2 protected mode.

See Also **ON** *event*, **PLAY** Function, **PLAY** Statements (Music)

Example See the **PLAY** function programming example, which uses the **PLAY ON** statement.

PLAY Statement (Music)

Action Plays music as specified by a string.

Syntax **PLAY** *commandstring$*

Remarks The *commandstring$* argument is a string expression that contains one or more music commands listed later in this entry.

The **PLAY** statement uses a concept similar to **DRAW** in that it embeds a music macro language (described below) in one statement. A set of commands, used as part of the **PLAY** statement, specifies a particular action.

The **VARPTR$**(*variablename*) form for variables must be used with **DRAW** and **PLAY** (music) statements in BASIC to execute substrings that contain variables. BASICA supports both the **VARPTR$** syntax and the syntax containing just the variable name. For example, consider these BASICA statements:

```
PLAY "XA$"
PLAY "O = I"
```

For BASIC programs, those statements should be written like this:

```
PLAY "X" + VARPTR$(A$)
PLAY "O=" + VARPTR$(I)
```

The *commandstring$* music commands are described as follows:

Octave command	Action
o *n*	Sets the current octave. There are seven octaves, numbered 0-6.
>	Increases octave by 1. Octave cannot go beyond 6.
<	Decreases octave by 1. Octave cannot drop below 0.

Tone command	Action
A–G	Plays a note in the range A–G. The number sign (#) or the plus sign (+) after a note specifies sharp; a minus sign (–) specifies flat.
N *n*	Plays note *n*. The range for n is 0–84 (in the seven possible octaves, there are 84 notes); an *n* value of 0 means a rest.

Duration command	Action
L *n*	Sets the length of each note. `L4` is a quarter note, `L1` is a whole note, etc. The range for *n* is 1–64. The length can also follow the note when a change of length only is desired for a particular note. For example, A16 can be equivalent to `L16A`.
MN	Sets "music normal" so that each note will play 7/8 of the time determined by the length (L).
ML	Sets "music legato" so that each note will play the full period set by length (L).
MS	Sets "music staccato" so that each note will play 3/4 of the time determined by the length (L).

Tempo command	Action
P *n*	Specifies a pause, ranging from 1 to 64. This option corresponds to the length of each note, set with L *n*.
T *n*	Sets the "tempo," or the number of L4 quarter notes in one minute. The range for *n* is 32–255. The default for *n* is 120. Because of the slow clock-interrupt rate, some notes will not play at higher tempos (`L64` at `T255`, for example).

Mode command	Action
MF	Sets music (**PLAY** statement) and **SOUND** to run in the foreground. That is, each subsequent note or sound will not start until the previous note or sound has finished. This is the default setting.
MB	Music (**PLAY** statement) and **SOUND** are set to run in the background. That is, each note or sound is placed in a buffer, allowing the BASIC program to continue executing while the note or sound plays in the background. The maximum number of notes that can be played in the background at one time is 32.

Suffix	**Action**
# or +	Follows a specified note and turns it into a sharp.
–	Follows a specified note and turns it into a flat.
.	A period after a note causes the note to play one-and-a-half times the length determined by L*T (length times tempo). The period has the same meaning as in a musical score. Multiple periods can appear after a note. Each period adds a length equal to one-half the length of the previous period. For example, the command `A.` plays 1 + 1/2, or 3/2 times the length; `A..` plays 1 + 1/2 + 1/4, or 7/4 times the length. Periods can appear after a pause (P). In this case, the pause length is scaled in the same way notes are scaled.

Substring command	**Action**
"X"+**VARPTR$**(*string*)	Executes a substring. This powerful command enables you to execute a second substring from a string. You can have one string expression execute another, which executes a third, and so on.

Note The **PLAY** statement is not available in OS/2 protected mode.

See Also **BEEP**, **ON** *event*, **PLAY** Function, **PLAY** Statements (Event Trapping), **SOUND**

Example See the **PLAY** function programming example, which uses the **PLAY** statement.

PMAP Function

Action Returns the window coordinate equivalent to a specified viewport coordinate, or the viewport coordinate equivalent to a specified window coordinate.

Syntax **PMAP** (*expression#*, *n%*)

Remarks The argument *expression#* specifies the known coordinate, either an x coordinate or y coordinate, in the window or in the viewport. The argument *n%* specifies the mapping function and can have one of the four following values:

If expression is	*n* must be	Returns:
Window x coordinate	0	Viewport x coordinate
Window y coordinate	1	Viewport y coordinate
Viewport x coordinate	2	Window x coordinate
Viewport y coordinate	3	Window y coordinate

The four **PMAP** functions allow you to find equivalent point locations between the window coordinates created with the **WINDOW** statement and the absolute screen coordinates or viewport coordinates as defined by the **VIEW** statement.

If no **VIEW** statement has been executed, or the most recently executed **VIEW** statement has no arguments, the viewport coordinates are equivalent to the absolute screen coordinates established by the most recently executed **SCREEN** statement.

See Also **POINT**, **VIEW**, **WINDOW**

Example See the **WINDOW** statement programming example, which uses the **PMAP** function.

POINT Function

Action Returns the current horizontal or vertical position of the graphics cursor, or the color of a specified pixel.

Syntax 1 **POINT** (*x%*,*y%*)

Syntax 2 **POINT** (*number%*)

Remarks The coordinates *x%* and *y%* refer to the viewport coordinates of the pixel being evaluated by the **POINT** function; **POINT** returns the color number of the indicated pixel. If the specified pixel is out of range, **POINT** returns the value –1.

POINT with the argument *number%* allows the user to retrieve the current graphics-cursor coordinates, as described in the following table:

Argument	Value returned
0	The current viewport x coordinate.
1	The current viewport y coordinate.
2	The current window x coordinate. This returns the same value as the `POINT(0)` function if the **WINDOW** statement has not been used.
3	The current window y coordinate. This returns the same value as the `POINT(1)` function if the **WINDOW** statement has not been used.

See Also **PMAP**, **SCREEN** Statement, **VIEW**, **WINDOW**

Example

The following example redraws an ellipse drawn with the **CIRCLE** statement, using **POINT** to find the border of the ellipse by testing for a change in color:

```
DEFINT X-Y
INPUT "Enter angle of tilt in degrees (0 to 90): ",Ang
SCREEN 1                                ' Medium resolution screen.
Ang = (3.1415926# / 180) * Ang          ' Convert degrees to radians.
Cs = COS(Ang) : Sn = SIN(Ang)
CIRCLE (45, 70), 50, 2, , , 2           ' Draw ellipse.
PAINT (45, 70), 2                       ' Paint interior of ellipse.
FOR Y = 20 TO 120
   FOR X = 20 TO 70
   ' Check each point in rectangle enclosing ellipse.
   IF POINT(X, Y) <> 0 THEN
       ' If the point is in the ellipse, plot a corresponding
       '  point in the "tilted" ellipse.
       Xnew = (X * Cs - Y * Sn) + 200 : Ynew = (X * Sn + Y * Cs)
       PSET(Xnew, Ynew), 2
   END IF
   NEXT
NEXT
END
```

POKE Statement

Action Writes a byte value into a specified memory location.

Syntax **POKE** *address*, *byte%*

Remarks The **POKE** statement uses the following arguments:

Argument	Description
address	An offset into the memory segment specified by the current default segment (as set by the **DEF SEG** statement). The value of *address* is between 0 and 65,535, inclusive.
byte%	The data byte to be written into the memory location. The argument *byte%* is an integer value between 0 and 255, inclusive. Any value for *byte%* that must be represented in more than one byte (for instance, a long integer or any real number) is accepted; however, only the low order byte is used. The value of any high order bytes is ignored.

If *address* is a single- or double-precision floating-point value, or a long integer, it is converted to a 2-byte integer.

The **POKE** statement complements the **PEEK** function.

Before using **POKE** to directly manipulate data stored in far memory, use the **DEF SEG** statement to set the current segment address. For example:

```
DEF SEG = SSEG(a$)      ' Set current segment address to address of a$.
Offset1 = SADD(a$)      ' Determine string's location within the segment.
' Write the byte stored in:
POKE Offset1, PEEK(Offset2)
```

Note BASIC moves string locations during run time. Therefore, the **DEF SEG** statement must be executed immediately before using the **POKE** statement.

POKE and Expanded Memory Arrays

Do not use **POKE** to manipulate expanded memory arrays. If you start QBX with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays less than 16K in size.
- Fixed-length string arrays less than 16K in size.
- User-defined-type arrays less than 16K in size.

If you want to use **POKE** to manipulate an array, first start QBX without the /Ea switch. (Without the /Ea switch, no arrays are stored in expanded memory.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started*.

Any address referred to by **POKE** in OS/2 protected mode must be open for writing. If **POKE** refers to a memory address for which your process does not have write permission, the operating system may generate a protection exception, or BASIC may generate the error message `Permission denied.`

Warning Use **POKE** carefully. If used incorrectly, it can cause BASIC or the operating system to fail.

See Also **DEF SEG**; **PEEK**; **VARPTR**, **VARSEG**; **VARPTR$**

Example See the **DEF SEG** statement programming example, which uses the **POKE** statement.

POS Function

Action Returns the current horizontal position (column) of the text cursor.

Syntax **POS**(*numeric-expression*)

Remarks If the screen cursor is in the leftmost column, this function returns a value of 1. To return the current vertical-line position of the cursor, use the **CSRLIN** function. The argument *numeric-expression* can be any numeric expression; while it passes no information, it is required to maintain consistent BASIC function syntax.

See Also **CSRLIN, LPOS**

Example The following example uses **POS** to start a new line after every 40 characters:

```
CLS                        ' Clear screen.
PRINT "This program starts a new line after every 40"
PRINT "characters you type. Press Ctrl+C to end."
PRINT
DO
   DO WHILE POS(0) < 41    ' Stay on same line until 40 characters
      DO                   ' printed.
         Char$ = INKEY$
      LOOP WHILE Char$ = ""
      ' If input is key combination CTRL-C then end; otherwise,
      ' print the character.
      IF ASC(Char$) = 3 THEN END ELSE PRINT Char$;
   LOOP
   PRINT                   ' Print a new line.
LOOP
```

PRESET Statement

Action Draws a specified point on the screen.

Syntax **PRESET** [[**STEP**]](*x%*,*y%*) [[,*color&*]]

Remarks The following list describes the parts of the **PRESET** statement:

Part	Description
STEP	Indicates that *x%* and *yc%* are relative, not absolute. The coordinates are treated as distances from the most recent graphics cursor location, not distances from the (0,0) screen coordinate. For example, if the most recent graphics cursor location were (10,10), `PRESET STEP (10,5)` would draw a point at (20,15).
x%	The x coordinate of the pixel that is to be set.
y%	The y coordinate of the pixel that is to be set.
color&	The color attribute for the specified pixel.

The valid range of screen coordinate values and color values depends on the screen mode established by the most recently executed **SCREEN** statement. If a coordinate is outside the current viewport, no action is taken, nor is an error message generated.

If you use **PRESET** without *color&*, the background color is selected. In contrast, if you use **PSET** without *color&*, the foreground color is selected. Otherwise, the two statements work exactly the same.

See Also **PSET**

Example The following example uses **PSET** and **PRESET** to draw a line 20 pixels long, then move the line across the screen from left to right:

```
SCREEN 1 : COLOR 1,1 : CLS
FOR I = 0 TO 299 STEP 3
   FOR J = I TO 20 + I
      PSET (J, 50), 2          ' Draw the line in new location.
   NEXT
   FOR J = I TO 20 + I
      PRESET (J, 50)           ' Erase the line.
   NEXT
NEXT
```

PRINT Statement

Action Outputs data to the screen.

Syntax **PRINT** [[*expressionlist*]] [[{;|,}]]

Remarks If *expressionlist* is omitted, a blank line is displayed. If *expressionlist* is included, the values of its expressions are printed on the screen.

The **PRINT** statement uses the following arguments:

Argument	Description
expressionlist	The values that are displayed on the screen. The argument *expressionlist* can contain numeric or string expressions. String literals must be enclosed in quotation marks.
{;|,}	Determines the screen location of the text cursor for the next screen input or output statement: a semicolon means the text cursor is placed immediately after the last character displayed; a comma means the text cursor is placed at the start of the next print zone.

The **PRINT** statement supports only elementary BASIC data types (integers, long integers, single-precision real numbers, double-precision real numbers, currency, and strings). To print information from a record, use individual record-element names in the **PRINT** statement, as in the following code fragment:

```
TYPE MyType
   Word AS STRING * 20
   Count AS LONG
END TYPE
DIM Myrec AS MyType

PRINT Myrec.Word
```

Item-Format Rules

A printed number always is followed by a space. If the number is positive, it also is preceded by a space; if the number is negative, it is preceded by a minus sign (–). If a single-precision number can be expressed as seven or fewer digits with no loss of accuracy, then it is printed in fixed-point format; otherwise, floating-point format is used. For example, the number 1.1E–6 is output displayed as .0000011, but the number 1.1E–7 is output as 1.1E–7.

If a double-precision number can be expressed with 15 or fewer digits and no loss of accuracy, it is printed in fixed-point format; otherwise, floating-point format is used. For example, the number 1.1D–14 is displayed as .000000000000011, but the number 1.1D–15 is output as 1.1D–15.

Print-Line Format Rules

The print line is divided into print zones of 14 spaces each. The position of each printed item is determined by the punctuation used to separate the items in *expressionlist*: a comma makes the next value print at the start of the next zone; a semicolon makes the next value print immediately after the last value. Using one or more spaces or tabs between expressions has the same effect as using a semicolon.

If a comma or a semicolon terminates the list of expressions, the next **PRINT** statement to execute prints on the same line, after spacing accordingly. If the expression list ends without a comma or a semicolon, a carriage-return-and-line-feed sequence is printed at the end of the line. If the printed line is wider than the screen width, BASIC goes to the next physical line and continues printing.

See Also

PRINT #, PRINT USING, WIDTH

Examples

The following examples show the use of commas and semicolons with **PRINT**.

First is an example of using commas in a **PRINT** statement to print each value at the beginning of the next print zone:

```
CLS                 ' Clear screen.
X = 5
PRINT X + 5, X - 5, X * (-5), X ^ 5
END
```

Output

```
10             0             -25            3125
```

In the second example, the semicolon at the end of the first **PRINT** statement makes the first two **PRINT** statements display output on the same line. The last **PRINT** statement prints a blank line before the next prompt.

```
CLS                   ' Clear screen.
DO
   INPUT "Input X (type 0 to quit): ", X
   IF X = 0 THEN
      EXIT DO
   ELSE
      PRINT X; "squared is"; X ^ 2; "and";
      PRINT X; "cubed is"; X^3
      PRINT
   END IF
LOOP
```

Output

```
Input X (type 0 to quit): 9
 9 squared is 81 and 9 cubed is 729

Input X (type 0 to quit): 21
 21 squared is 441 and 21 cubed is 9261

Input X (type 0 to quit): 0
```

In the third example, the semicolons in the **PRINT** statement print each value immediately after the preceding value. Note that a space always follows a number and precedes a positive number.

```
CLS                   ' Clear screen.
FOR X = 1 TO 5
  J = J + 5
  K = K + 10
  PRINT J; K;
NEXT X
```

Output

```
5  10  10  20  15  30  20  40  25  50
```

PRINT # Statement

Action Writes data to a sequential device or file.

Syntax **PRINT** #*filenumber%*, [[**USING** *formatstring$*;]] *expressionlist*[[{,|;}]]

Remarks The **PRINT #** statement uses the following arguments:

Argument	Description
filenumber%	The number of an open sequential file.
formatstring$	Specifies the exact format in which values are written to the file; described under **PRINT USING**.
expressionlist	Numeric or string expressions to be written to the file.

Spaces, commas, and semicolons in *expressionlist* have the same meaning as they have in a **PRINT** statement.

If you omit *expressionlist*, the **PRINT #** statement prints a blank line in the file.

PRINT # works like **PRINT** and writes an image of the data to the file, just as the data would be displayed on the terminal screen. For this reason, be careful to delimit the data so it is output correctly. If you use commas as delimiters, the blanks between print fields are also written to the file. For more information, see "File and Device I/O" in the *Programmer's Guide*.

See Also **OPEN** (File I/O), **PRINT**, **PRINT USING**, **WRITE #**

Example The following example shows the effects of using data delimiters with **PRINT #**:

```
CONST A$ = "CAMERA, AUTOFOCUS", B$ = "September 20, 1989", C$ = "42"
Q$ = CHR$(34)
OPEN "INVENT.DAT" FOR OUTPUT AS #1   ' Open INVENT.DAT for writing.
PRINT #1, A$; B$; C$                 ' Write without delimiters.
PRINT #1, Q$; A$; Q$; Q$; B$; Q$; Q$; C$; Q$      ' With delimiters.
CLOSE #1
OPEN "INVENT.DAT" FOR INPUT AS #1    ' Open INVENT.DAT for reading.
FOR I% = 1 TO 2                      ' Read first two records and print.
   INPUT #1, First$, Second$, Third$
   PRINT First$; TAB(30); Second$; TAB(60); Third$
NEXT
CLOSE #1
```

Output

```
CAMERA                       AUTOFOCUSSeptember 20         198942
CAMERA, AUTOFOCUS            September 20, 1989            42
```

PRINT USING Statement

Action Prints strings or numbers using a specified format.

Syntax **PRINT USING** *formatstring$*; *expressionlist* ⟦{;|,}⟧

Remarks The **PRINT USING** statement uses the following arguments:

Argument	Description
formatstring$	A string literal (or variable) that contains literal characters to print (such as labels) and special formatting characters. The latter determine the field and format of printed strings or numbers.
expressionlist	The values displayed on the screen. Commas, semicolons, spaces, or tabs can be used in *expressionlist* to separate items. In contrast with the **PRINT** statement, delimiters in the *expressionlist* argument used with **PRINT USING** have no effect on item placement.
{;|,}	Determines the screen location of the text cursor for the next screen input or output statement: a semicolon means the text cursor is placed immediately after the last character displayed; a comma means the text cursor is placed at the start of the next print zone.

When **PRINT USING** is used to print strings, you can use one of three formatting characters to format the string field, as described in the following list:

Character	Rules
!	Only the first character in the given string is to be printed.
\\	Prints 2 + *n* characters from the string, where *n* is the number of spaces between the two backslashes. For example, if the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.
&	The string is output without modification.

When **PRINT USING** is used to print numbers, the following special characters can be used to format the numeric field:

Character	Rules
#	Represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.
.	Prints a decimal point. A decimal point can be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit is always printed (as 0, if necessary). Numbers are rounded as necessary.
+	Prints the sign of the number (+ or –) before the number (if it appears at the beginning of the format string) or after (if it appears at the end).
–	Prints a negative number with a trailing minus sign if it appears at the end of the format string.
**	Fills leading spaces in the numeric field with asterisks. Also specifies positions for two more digits.
$$	Prints a dollar sign to the immediate left of the formatted number. Specifies two more digit positions, one of which is the dollar sign.
**$	Combines effects of double-asterisk and double-dollar-sign symbols. Leading spaces are filled with asterisks and a dollar sign is printed before the number. Specifies three more digit positions, one of which is the dollar sign. When negative numbers are printed, the minus sign appears to the immediate left of the dollar sign.
,	If the comma appears to the left of the decimal point in a format string, it makes a comma print to the left of every third digit left of the decimal point. If the comma appears at the end of the format string, it is printed as part of the string. Specifies another digit position. Has no effect if used with exponential (^^^^ or ^^^^^) format.
^^^^	Specifies exponential format. Five carets (^^^^^) allows D+*xxx* to be printed for larger numbers. Any decimal point position can be specified. The significant digits are left-justified and the exponent is adjusted. Unless a leading +, trailing +, or – is specified, one digit position is used to the left of the decimal point to print a space or a minus sign.
_	An underscore in the format string prints the next character as a literal character. A literal underscore is printed as the result of two underscores (_ _) in the format string.

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number. If the number of digits specified exceeds 24, BASIC generates the error message `Illegal function call.`

Examples

The following examples show the use of string- and numeric-formatting characters with **PRINT USING**.

This is an example of using string-formatting characters:

```
CLS     ' Clear screen.
A$ = "LOOK" : B$ = "OUT"
PRINT USING "!"; A$; B$              ' First characters of A$ and B$.
PRINT USING "\  \"; A$; B$           ' Two spaces between backslashes,
                                     ' prints four characters from A$;
PRINT USING "\   \"; A$; B$; "!!"    ' three spaces prints A$ and
                                     ' a blank.
PRINT USING "!"; A$;                 ' First character from A$ and
PRINT USING "&"; B$                  ' all of B$ on one line.
```

Output

```
LO
LOOKOUT
LOOK OUT  !!
LOUT
```

This example shows the effects of different combinations of numeric formatting characters:

```
' Format and print numeric data.
CLS    ' Clear screen.
PRINT USING "##.##"; .78
PRINT USING "###.##"; 987.654
PRINT USING "##.##   "; 10.2, 5.3, 66.789, .234
PRINT USING "+##.##   "; -68.95, 2.4, 55.6, -.9
PRINT USING "##.##-   "; -68.95, 22.449, -7.01
PRINT USING "**#.#   "; 12.39, -0.9, 765.1
PRINT USING "$$###.##"; 456.78
PRINT USING "**$##.##"; 2.34
PRINT USING "####,.##"; 1234.5
PRINT USING "##.##^^^^"; 234.56
PRINT USING ".####^^^^-"; -888888
PRINT USING "+.##^^^^^"; 123
PRINT USING "_!##.##_!"; 12.34
PRINT USING "##.##"; 111.22
PRINT USING ".##"; .999
```

Output

```
 0.78
987.65
10.20    5.30   66.79    0.23
-68.95    +2.40   +55.60    -0.90
68.95-   22.45     7.01-
*12.4   *-0.9   765.1
 $456.78
***$2.34
1,234.50
 2.35E+02
.8889E+06-
+.12E+003
!12.34!
%111.22
%1.00
```

PSET Statement

Action Draws a specified point on the screen.

Syntax **PSET** [[**STEP**]](*x%*,*y%*) [[,*color&*]]

Remarks The following list describes the parts of the **PSET** statement:

Part	Description
STEP	Indicates that *x%* and *y%* are relative, not absolute. Coordinates are treated as distances from the most recent graphics cursor location, not distances from the (0,0) screen coordinate. For example, if the most recent graphics cursor location were (10,10), `PSET STEP (10,5)` would refer to the point at (20,15).
x%	The x coordinate of the pixel that is to be set.
y%	The y coordinate of the pixel that is to be set.
color&	The color attribute for the specified pixel.

The valid range of screen coordinate values and color values depends on the screen mode established by the most recently executed **SCREEN** statement. If a coordinate is outside the current viewport, no action is taken, nor is an error message generated.

If you use **PSET** without specifying *color&*, the foreground color is selected. In contrast, if you use **PRESET** without specifying *color&*, the background color is selected. Otherwise, the two statements work exactly the same.

See Also **PRESET**

Example The following example draws a line from (0,0) to (100,100), then erases the line by writing over it with the background color:

```
SCREEN 2
FOR I = 0 TO 100
   PSET (I, I)
NEXT I
LOCATE 16, 2: INPUT "Press the Return key to erase the line ", Gar$
' Now erase the line.
PSET (I - 1, I - 1), 0
FOR I = 0 TO 100
   PSET STEP(-1, -1), 0
NEXT I
LOCATE 16, 2: PRINT "                                          "
```

PUT Statement (File I/O)

Action Writes from a variable or a random-access buffer to a file.

Syntax **PUT** [[#]]*filenumber%*[[, [[*recordnumber%*]][[, *variable*]]]]

Remarks Do not use **PUT** on ISAM files.

The **PUT** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
recordnumber%	For random-access files, the number of the record to be written. For binary-mode files, the byte position where writing is done. The first record or byte position in a file is 1. If you omit *recordnumber%*, the next record or byte (the one after the last **GET** or **PUT** statement, or the one pointed to by the last **SEEK**) is written to. The largest possible record number is $2^{31}-1$ or 2,147,483,647.
variable	A variable that contains output to be written to the file. The **PUT** statement writes as many bytes to the file as there are bytes in the variable.

If you specify a variable, you do not need to use **MKI$**, **MKL$**, **MKS$**, **MKD$**, or **MKC$** to convert numeric fields before writing. You cannot use a **FIELD** statement with the file if you use the variable argument.

For random-access files, you can use any variable as long as the length of the variable is less than or equal to the length of the record. Usually, a record variable defined to match the fields in a data record is used. A record cannot be longer than 32,767 bytes.

For binary-mode files, you can use any variable.

When you use a variable-length string variable, the statement writes as many bytes as there are characters in the string's value. For example, the following two statements write 15 bytes to file number 1:

```
VarString$=STRING$(15, "X")
PUT #1,,VarString$
```

See the examples for more information about using variables rather than **FIELD** statements for random-access files.

You can omit *recordnumber%*, *variable*, or both. If you omit *recordnumber%* but include a variable, you must still include the commas:

```
PUT #4,,FileBuffer
```

If you omit both arguments, do not include the commas:

```
PUT #4
```

GET and **PUT** statements allow fixed-length input and output for BASIC communication files. Be careful using **GET** and **PUT** for communication because **PUT** writes a fixed number of characters and may wait indefinitely if there is a communication failure.

Note When using a file buffer defined by a **FIELD** statement, **LSET**, **RSET**, **PRINT #** , **PRINT # USING**, and **WRITE #** can be used to put characters in the random-file buffer before executing a **PUT** statement. In the case of **WRITE #**, BASIC pads the buffer with spaces up to the carriage return. If you attempt to read or write past the end of the buffer, BASIC generates the error message `FIELD overflow`.

See Also **CVI**, **CVL**, **CVS**, **CVD**, **CVC**; **GET** (File I/O); **LSET**; **MKI$**, **MKL$**, **MKS$**, **MKD$**, **MKC$**; **OPEN** (File I/O)

Example The following example reads names and test scores from the console and stores them in a random-access file:

```
' Define record fields.
TYPE TestRecord
   NameField  AS STRING * 20
   ScoreField AS SINGLE
END TYPE
DIM FileBuffer AS TestRecord
DIM I AS LONG
' Open the test data file.
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 LEN = LEN(FileBuffer)
' Read pairs of names and scores from the console.
CLS    ' Clear screen.
I = 0
DO
    I = I + 1
    INPUT "Name ? ", FileBuffer.NameField
    INPUT "Score? ", FileBuffer.ScoreField
    INPUT "-->More (y/n)? ", Resp$
    PUT #1, I, FileBuffer
LOOP UNTIL UCASE$(MID$(Resp$, 1, 1)) = "N"
PRINT I; " records written."
CLOSE #1
KILL "TESTDAT.DAT"     ' Remove file from disk.
```

PUT Statement (Graphics)

Action Places a graphic image obtained by a **GET** statement onto the screen.

Syntax **PUT** [[**STEP**]] (*x!*, *y!*),*arrayname#* [[(*indexes%*)]] [[,*actionverb*]]

Remarks The list below describes the parts of the **PUT** statement:

Part	Description
STEP	Keyword indicating that the given x and y coordinates are relative to the most recently plotted point. The coordinates are treated as distances from the most-recent cursor location, not distances from the (0,0) screen coordinate. For example, if the most recent cursor location were (10,10) then the following statement would put the object stored in `Ball` at (20,15): `PUT STEP (10,5),Ball`
(*x!*,*y!*)	Coordinates that specify the top-left corner of the rectangle enclosing the image to be placed in the current output window. The entire rectangle to be put on the screen must be within the bounds of the current viewport. Note that if a **WINDOW** statement without the argument **SCREEN** appears in a program before **PUT**, the coordinates refer to the lower-left corner of the rectangle.
arrayname#	The name of the array that holds the image. See the **GET** statement for information about the number of elements that are required in the array, which can be multidimensional.
indexes%	Specifies that the image is retrieved starting from the designated array element, rather than at the first array element.
actionverb	Determines interaction between stored image and the one already on the screen. This allows display of the image with special effects.

The different values for *actionverb* are described in the following list. The default *actionverb* value is **XOR**.

Verb	Description
XOR	Causes the points on the screen to be inverted where a point exists in the array image. When an image is placed on the screen against a complex background twice, the background is restored. This behavior is exactly like that of the cursor. You can move an object around the screen without erasing the background, thus creating animation effects.

PSET — Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen with **GET**. The **PSET** argument draws the image as stored, wiping out any existing image.

PRESET — The same as **PSET** except that a negative image (for example, black on white) is produced.

AND — Used when the image is to be transferred over an existing image on the screen. The resulting merged image is the result of a logical-**AND** operation on the stored image and the screen. Points that had the same color in both the existing image and the stored image remain the same color. Points that do not have the same color in both the existing image and the stored image do not.

OR — Used to superimpose the image onto an existing image. The resulting image is the product of a logical-**OR** operation on the stored image and the screen image. The stored image does not erase the previous screen contents.

See Also

GET (Graphics)

Example

The following example creates a moving white ball that ricochets off the sides of the screen until you press a key:

```
DEFINT A-Z
DIM Ball(84)           ' Set the dimensions for an integer array large
                       ' enough to hold ball.
SCREEN 2               ' 640 pixels by 200 pixels screen resolution.

INPUT "Press any key to end; press <ENTER> to start", Test$
CLS
CIRCLE (16, 16), 14 ' Draw and paint ball.
PAINT (16, 16), 1
GET (0, 0)-(32, 32), Ball
X = 0 : Y = 0
Xdelta = 2 : Ydelta = 1
```

```
DO
   ' Continue moving in same direction as long as ball is within
   ' the boundaries of the screen - (0,0) to (640,200).
   X = X + Xdelta : Y = Y + Ydelta
   IF INKEY$ <> "" THEN END  ' Test for key press.
   ' Change X direction if ball hits left or right edge.
   IF (X < 1 OR X > 600) THEN
      Xdelta = -Xdelta
      BEEP
   END IF
   ' Change Y direction if ball hits top or bottom edge.
   IF (Y < 1 OR Y > 160) THEN
      Ydelta = -Ydelta
      BEEP
   END IF
   ' Put new image on screen, simultaneously erasing old image.
   PUT (X, Y), Ball, PSET
LOOP
END
```

RANDOMIZE Statement

Action Initializes (reseeds) the random-number generator.

Syntax **RANDOMIZE** ⟦*expression%*⟧

Remarks If you omit *expression%*, BASIC pauses and asks for a value by printing the following messages before executing the **RANDOMIZE** statement:

```
Random Number Seed (-32768 to 32767)?
```

When you use the argument *expression%*, BASIC uses this value to initialize the random-number generator.

If the random-number generator is not reseeded, the **RND** function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a **RANDOMIZE** statement at the beginning of the program and change the argument with each run.

A convenient way to initialize the random-number generator is to use the **TIMER** function. Using **TIMER** ensures a new series of random numbers each time you use the program. See the example below.

See Also **RND, TIMER**

Example The following example uses **RANDOMIZE** to seed and reseed the random number generator. It uses the **RND** function to simulate rolling a pair of dice.

```
' Use the timer as the seed for the number generator.
RANDOMIZE TIMER
DO
   ' Simulate rolling two dice using RND.
   D1 = INT(RND * 6) + 1
   D2 = INT(RND * 6) + 1
   ' Report the roll.
   CLS    ' Clear screen.
   PRINT "You rolled a"; D1; "and a"; D2; "for a total of"; D1 + D2
   INPUT "Roll again (Y/N)"; Resp$
   PRINT
LOOP UNTIL UCASE$(MID$(Resp$, 1, 1)) = "N"
END
```

Output

```
You rolled a 3 and a 5 for a total of 8
Roll again (Y/N)? n
```

READ Statement

Action Reads values from a **DATA** statement and assigns the values to variables.

Syntax **READ** *variablelist*

Remarks The argument *variablelist* is a series of BASIC variables that receive the data from a **DATA** statement. The variables are separated by commas and can be string or numeric. Only individual elements of a record variable can appear in a **READ** statement.

The following table describes what happens when you try to read data of one data type into a variable with a different data type:

If you try to read this:	*Into this:*	*The result is:*
String value	Numeric variable	A run-time error.
Numeric value	String variable	The value is stored as a string of numerals (no error is produced).
Any numeric value	Integer variables	The value is rounded before it is assigned to the variable.
Numeric value	A variable not large enough to handle the numeric variable	A run-time error.
String value	Fixed-length string variables	Truncated if the string is too long; left-justified and padded with blanks if the string is shorter than the variable.

Each variable in a **READ** statement receives its value from some **DATA** statement. Which value the variable receives depends on how many values have previously been read. The values of all **DATA** statements in a module can be considered as a single list of values. Each value in this list is assigned in turn to the variables specified in **READ** statements. It doesn't matter how many values are specified in a given **DATA** statement or how many variables are specified in a **READ** statement. If you attempt to read more values than are specified in all of the statements combined, BASIC generates the error message `Out of data.`

Use the **RESTORE** statement to reread **DATA** statements.

See Also **DATA, RESTORE**

Example

The following example shows how you can use a **READ** statement to assign values to the user-defined type Employee.

```
TYPE Employee
   EmpName AS STRING * 35
   SocSec AS STRING * 11
   JobClass AS INTEGER
END TYPE

CLS                ' Clear screen.
DIM ThisEmp AS Employee
DATA "Julia Magruder","300-32-3403",3
DATA "Amelie Reeves Troubetzkoy","777-29-3206",7

' Read first data input line and verify by printing data.
READ ThisEmp.EmpName, ThisEmp.SocSec, ThisEmp.JobClass
PRINT "Employee is "; ThisEmp.EmpName
PRINT "Employee's social security number is "; ThisEmp.SocSec
PRINT "Employee's job class is"; ThisEmp.JobClass
PRINT  ' Print blank line

' Read second data input line and verify.
READ ThisEmp.EmpName, ThisEmp.SocSec, ThisEmp.JobClass
PRINT "Employee is "; ThisEmp.EmpName
PRINT "Employee's social security number is "; ThisEmp.SocSec
PRINT "Employee's job class is"; ThisEmp.JobClass
```

Output

```
Employee is Julia Magruder
Employee's social security number is 300-32-3403
Employee's job class is 3

Employee is Amelie Reeves Troubetzkoy
Employee's social security number is 777-29-3206
Employee's job class is 7
```

REDIM Statement

Action Changes the space allocated to an array that has been declared dynamic.

Syntax **REDIM** [[**SHARED**]] *variable*(*subscripts*) [[**AS** *type*]] [[, *variable*(*subscripts*) [[**AS** *type*]]]]...

Remarks The **REDIM** statement uses the following arguments:

Arguments	Description
SHARED	The optional **SHARED** attribute allows a module to share variables with all the procedures in the module; this differs from the **SHARED** statement, which affects only the variables within a single module. **SHARED** can be used in **REDIM** statements only in the module-level code.
variable	A BASIC variable name.
subscripts	The dimensions of the array. Multiple dimensions can be declared. The subscript syntax is described below.
AS *type*	Declares the type of the variable. The type can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING** (for variable-length strings), **STRING** * *length* (for fixed-length strings), **CURRENCY**, or a user-defined type.

Subscripts in **REDIM** have the following form:

[[*lower* **TO**]] *upper* [[, [[*lower* **TO**]] *upper*]]...

The **TO** keyword provides a way to indicate both the lower and the upper bounds of an array's subscripts. The arguments *lower* and *upper* are numeric expressions that specify the lowest and highest value for the subscript. For more information about using the **TO** keyword, see the **DIM** statement. For more information about static and dynamic arrays, see Appendix B, "Data Types, Constants, Variables, and Arrays" in the *Programmer's Guide*.

When a **REDIM** statement is compiled, all arrays declared in the statement are treated as dynamic. At run time, when a **REDIM** statement is executed, the array is deallocated (if it is already allocated) and then reallocated with the new dimensions. Old array-element values are lost because all numeric elements are reset to 0, and all string elements are reset to null strings.

Although you can change the *size* of an array's dimensions with the **REDIM** statement, you can not change the number of dimensions. For example, the following statements are legal:

```
' $DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(20,15) ' Array A still has two dimensions.
```

However, the following statements are *not* legal, and if you use them, BASIC generates the error message `Wrong number of dimensions:`

```
' $DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(5,5,5) ' Changed number of dimensions from two to three.
```

Note

BASIC now supports the **CURRENCY** data type (type suffix @). This is used in the **AS** *type* clause of **REDIM.** BASIC now supports static arrays in user-defined types.

See Also

DIM, ERASE

Example

The following example shows how to use **REDIM** to allocate an array of records and then how to free the memory that the records use.

```
TYPE KeyElement
   Word AS STRING * 20
   Count AS INTEGER
END TYPE

' Make arrays dynamic.
' $DYNAMIC
CLS                           ' Clear screen.
' Allocate an array of records when you need it.
REDIM Keywords(100) AS KeyElement
Keywords(99).Word = "ERASE"
Keywords(99).Count = 2
PRINT "Keyword 99 is "; Keywords(99).Word
PRINT "Count is"; Keywords(99).Count
' Free the space taken by Keywords when you're finished.
ERASE Keywords
END
```

Output

```
Keyword 99 is ERASE
Count is 2
```

REM Statement

Action Allows explanatory remarks to be inserted in a program.

Syntax 1 **REM** *remark*

Syntax 2 ' *remark*

Remarks **REM** statements are not compiled, but they appear exactly as entered when the program is listed. You can branch from a **GOTO** or **GOSUB** statement to a **REM** statement. Execution continues with the first executable statement after the **REM** statement.

A single quotation mark can be used instead of the **REM** keyword. If the **REM** keyword follows other statements on a line, it must be separated from the statements by a colon.

REM statements also are used to introduce metacommands. For more information, see Chapter 2, "SUB and FUNCTION Procedures" in the *Programmer's Guide.*

Note Do not use the single quotation form of the **REM** statement in a **DATA** statement because it will be considered valid data.

Examples The following example shows two equivalent remark statements. Note that you must precede a REM statement at the end of a line with a colon.

This is not a complete program.

```
DIM Array(23)
FOR I = 1 TO 23 : Array(I) = 1 : NEXT I : REM Initialize the array.
FOR I = 1 TO 23 : Array(I) = 1 : NEXT I   ' Initialize the array.
```

RESET Statement

Action Closes all disk files.

Syntax **RESET**

Remarks The **RESET** statement closes all open disk files and writes data still in the file buffers to disk.

See Also **CLOSE, END, SYSTEM**

Example The following example opens several files for sequential output, then performs a **RESET**. The program attempts to write to the previously opened files. The error that occurs is trapped and a message is printed indicating that all files have been closed using the **RESET** statement.

```
DEFINT A-Z
ON ERROR GOTO ErrHandler     ' Set up the error-handling routine.

CLS
FOR I = 1 TO 3
    OPEN "Test" + RIGHT$(STR$(I), 1) + ".dat" FOR OUTPUT AS FREEFILE
    PRINT "File #"; I; "has been opened for output."
NEXT I
PRINT : PRINT "Press any key to reset all open files."
PRINT
Z$ = INPUT$(1)
RESET
FOR I = 1 TO 3
    PRINT "Trying to write to file #"; I
    PRINT #I, "Test data"
NEXT I
END

ErrHandler:
' Error 52 is "Bad file name or number"
IF ERR = 52 THEN PRINT "  File #"; I; "not open. RESET closed it."
RESUME NEXT
```

RESTORE Statement

Action Allows **DATA** statements to be reread from a specified line.

Syntax **RESTORE** [[{*linelabel* | *linenumber*}]]

Remarks After executing a **RESTORE** statement without a specified *linelabel* or *linenumber*, the next **READ** statement gets the first item in the first **DATA** statement in the program.

If *linelabel* or *linenumber* is specified, the next **READ** statement gets the first item in the specified **DATA** statement. If a line is specified, the line label or line number must be in the module-level code. (Note that in the QBX environment, **DATA** statements are automatically moved to the module-level code.)

See Also **DATA, READ**

Example See the **SEEK** (file I/O) statement programming example, which uses the **RESTORE** statement.

RESUME Statement

Action Resumes program execution after an error-handling routine is finished.

Syntax **RESUME** {[[**0**]] | **NEXT** | *line*}

Remarks The different forms of the **RESUME** statement redirect program flow as described in the following list:

Part	Description
RESUME[[**0**]]	Program execution resumes with the statement that caused the error, or at the most recent call out of the error-handling procedure or module.
RESUME NEXT	Execution resumes with the statement immediately following the one that caused the error, or with the statement immediately following the most recent call out of the error-handling procedure or module.
line	Execution resumes at *line*, a label or a line number. The argument *line* must be in the same procedure (for local error-handlers) or within the same module (for module-level error-handlers).

The location where execution resumes is based on the location of the error handler in which the error is trapped, not necessarily on the location where the error occurred.

The following table summarizes the resumption rules for the **RESUME** [[**0**]] statement:

Error handler	Location of error	Where program resumes
Local	Same procedure	Statement that caused error
Module level	Same module	Statement that caused error
Local or module level	Another procedure, same or another module	Statement that last called out of the procedure or module that contains the error handler

Note If BASIC had to search for the error handler (the error handler that contains the **RESUME** statement is in a procedure or module other than the one in which the the error occurred), then the last statement executed in that procedure (or module) is the last call out of that procedure or module.

As a rule, avoid using a *line* argument with a **RESUME** statement in a module-level error handler, unless you expect errors to occur only at the module level.

If you use a **RESUME** statement outside an error-handling routine, BASIC generates the error message `RESUME without error.`

When an error handling routine is active and the end of the program text is encountered before executing a **RESUME** statement, BASIC generates the error message `No RESUME`. This also is true if an **END** statement (or an **END SUB**, **END FUNCTION**, or **END DEF** statement for a local error handler) is executed before a **RESUME**.

Note

Programs containing error-handling routines must be compiled with either the /E (On error) or /X (Resume next) options when you are compiling from the BASIC command line.

See Also **ON ERROR**

Example See the **ON ERROR** statement programming example, which uses the **RESUME** statement.

RETRIEVE Statement

Action Fetches the current record in an ISAM table and places its data into a record variable.

Syntax **RETRIEVE** [[#]]*filenumber%*,*recordvariable*

Remarks **RETRIEVE** places the current record's data into *recordvariable*. You can change the data in *recordvariable*, then update the current record with the changes you've made. Use the **UPDATE** statement to update the current record.

The **RETRIEVE** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the table.
recordvariable	The name of the variable that will hold the current record's data. It is a variable of the user-defined type *tabletype* that was specified in the **OPEN** statement.

RETRIEVE has no effect on the current position.

If the values passed to *recordvariable* do not match the record strucure in the user-defined type, BASIC generates the error message `Type Mismatch.` The record structure includes the names and types of columns or fields.

See Also **INSERT**, **UPDATE**

Example See the programming example for the **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements, which uses the **RETRIEVE** statement.

RETURN Statement

Action Returns control from a routine.

Syntax **RETURN** [[{*linelabel* | *linenumber*}]]

Remarks Without a line label or number, **RETURN** continues execution where an event occurred (for event handling) or at the statement following the **GOSUB** statement (for subroutine calls). **GOSUB** and **RETURN** without a line number can be used anywhere in a program, but the **GOSUB** and corresponding **RETURN** must be at the same level.

The *linelabel* or *linenumber* in the **RETURN** statement causes an unconditional return from a **GOSUB** subroutine to the specified line. **RETURN** with a line label or line number can return control to a statement in the module-level code only, not in procedure-level code.

A **RETURN** statement cannot be used to return control to a calling program from a **SUB** procedure. Use **EXIT SUB** for this purpose.

Note BASIC's **SUB** procedures provide a better-structured alternative to **GOSUB** subroutines.

See Also **EXIT**, **GOSUB**, **ON** *event*

Example See the **GOSUB** statement programming example, which uses the **RETURN** statement.

RIGHT$ Function

Action Returns a string consisting of the rightmost *n%* characters of a string.

Syntax **RIGHT$**(*stringexpression$*, *n%*)

Remarks The argument *stringexpression$* can be any string variable, string constant, or string expression.

The argument *n%* is a numeric expression between 0 and 32,767, inclusive, indicating how many characters are to be returned.

If *n%* is 0, the null string (length 0) is returned.

If *n%* is greater than or equal to the number of characters in *stringexpression$*, the entire string is returned. To find the number of characters in *stringexpression$*, use **LEN**(*stringexpression$*).

See Also **LEFT$**, **MID$** Function

Example The following example converts names entered in the form "Firstname [Middlename] Lastname" to the form "Lastname, Firstname [Middlename]."

```
CLS                         ' Clear screen.
LINE INPUT "Name: "; Nm$
I = 1 : Sppos = 0
DO WHILE I > 0
    I = INSTR(Sppos + 1, Nm$, " ")   ' Get position of next space.
    IF I > 0 THEN Sppos = I
LOOP

' Sppos now points to the position of the last space.
IF Sppos = 0 THEN
    PRINT Nm$               ' Only a last name was input.
ELSE
    ' Everything after last space.
    Lastname$ = RIGHT$(Nm$, LEN(Nm$) - Sppos)
    ' Everything before last space.
    Firstname$ = LEFT$(Nm$, Sppos - 1)
    PRINT Lastname$ ", " Firstname$
END IF
END
```

RMDIR Statement

Action Removes an existing directory.

Syntax **RMDIR** *pathname$*

Remarks The *pathname$* is the name of the directory that is to be deleted. The *pathname$* must be a string of fewer than 64 characters. The directory to be removed must be empty except for the working directory ('.') and the parent directory ('..'); otherwise, BASIC generates one of two error messages, either `Path not found` or `Path/File access error`.

RMDIR works like the DOS command of the same name. However, the syntax in BASIC cannot be shortened to RD, as it can in DOS.

See Also **CHDIR**, **MKDIR**

Example The following example shows how to use **RMDIR** to remove a subdirectory.

```
CHDIR "C:\SALES\TEMP"   ' Move to \TEMP subdirectory in \SALES.
KILL "*.*"              ' Remove all files in \TEMP.
CHDIR ".."              ' Move back up to \SALES.
RMDIR "TEMP"            ' Remove \TEMP subdirectory.
```

RND Function

Action Returns a single-precision random number between 0 and 1.

Syntax **RND** [[(*n#*)]]

Remarks The value of *n#* determines how **RND** generates the next random number:

Value	Number returned
n# < 0	Always returns the same number for any given *n#*.
n# > 0 or *n* omitted	Returns the next random number in the sequence.
n# = 0	Returns the last number generated.

Even if *n#* > 0, the same sequence of random numbers is generated each time the program is run unless you initialize the random-number generator each time. (See the **RANDOMIZE** statement entry for more information about initializing the random-number generator.)

To produce random *integers* in a given range, use this formula:

INT ((*upperbound* – *lowerbound* + 1) * **RND** + *lowerbound*)

In this formula, *upperbound* is the highest number in the range, and *lowerbound* is the lowest number in the range.

Example See the **RANDOMIZE** statement programming example, which uses the **RND** function.

ROLLBACK, ROLLBACK ALL Statements

Action Rescind all or part of the operations of a transaction (a series of ISAM database operations).

Syntax **ROLLBACK** [[*savepoint%*]]
ROLLBACK ALL

Remarks The *savepoint%* is an integer that identifies a savepoint within a transaction—a series of ISAM database operations that is either committed as a whole or rescinded. Use the **SAVEPOINT** function to designate a savepoint, which marks the beginning of a subset of operations that can be rescinded.

If you specify a *savepoint%* argument, **ROLLBACK** returns the data affected by a transaction to its state at that savepoint.

If you do not specify a *savepoint%* argument, **ROLLBACK** returns the data affected by the transaction to its state at the previous savepoint, or at the beginning of the transaction if there are no intermediate savepoints.

ROLLBACK ALL rescinds all operations in a transaction and returns the data to its initial state at the beginning of the transaction. Use **BEGINTRANS** to indicate the beginning of a transaction. Use **COMMITTRANS** to commit all operations since the most recent **BEGINTRANS** statement.

See Also **BEGINTRANS, COMMITTRANS, SAVEPOINT**

Example See the **BEGINTRANS** statement programming example, which uses the **ROLLBACK** statement.

RSET Statement

Action Moves data from memory to a random-access file buffer (in preparation for a **PUT** statement), or right-justifies the value of a string in a string variable.

Syntax **RSET** *stringvariable$=stringexpression$*

Remarks The **RSET** statement uses the following arguments:

Argument	Description
stringvariable$	Usually a random-access file field defined in a **FIELD** statement, although it can be any string variable.
stringexpression$	The value that is assigned to *stringvariable$* and is right-justified.

If *stringexpression$* requires fewer bytes than were defined for *stringvariable$* in the **FIELD** statement, the **RSET** statement right-justifies the string in the field (**LSET** left-justifies the string). Spaces are used to pad the extra positions. If the string is too long for the field, both **LSET** and **RSET** truncate characters from the right. Numeric values must be converted to strings before they are justified with the **RSET** or **LSET** statements.

The **RSET** statement can be used with string variables unrelated to **FIELD** statements. When used with a fixed-length string variable, the value is right-justified and left-padded with blanks.

When **RSET** is used with a variable-length string, the string is treated as a fixed field. The length of the field is the length of the value the variable had before the **RSET** statement.

See Also **FIELD**; **LSET**; **MKI$**, **MKL$**, **MKS$**, **MKD$**, **MKC$**; **PUT** (File I/O)

Example See the **LSET** statement programming example, which uses the **RSET** statement.

RTRIM$ Function

Action Returns a string with trailing (rightmost) spaces removed.

Syntax **RTRIM$**(*stringexpression$*)

Remarks The *stringexpression$* can be any string expression. The **RTRIM$** function works with both fixed- and variable-length string variables.

See Also **LTRIM$**

Example The following example shows the effects of **RTRIM$** on fixed- and variable-length strings:

```
DIM FixStr AS STRING * 10
CLS    ' Clear screen.
PRINT "         1         2"
PRINT "12345678901234567890"
FixStr = "Twine"
PRINT FixStr + "*"
PRINT RTRIM$(FixStr) + "*"
VarStr$ = "Braided" + SPACE$(10)
PRINT VarStr$ + "*"
PRINT RTRIM$(VarStr$) + "*"
```

Output

```
         1         2
12345678901234567890
Twine     *
Twine*
Braided          *
Braided*
```

RUN Statement

Action Restarts the program currently in memory, or executes a specified program.

Syntax **RUN** [[{*linenumber* | *filespec$*}]]

Remarks The **RUN** statement uses the following arguments:

Argument	Description
linenumber	The numeric label of the line where execution begins. If no argument is given, execution begins at the first executable line of code.
filespec$	A string expression that names the program file to load and run. The current program is cleared from memory before the specified program is loaded.

The line where execution begins must be in the module-level code. Therefore, a **RUN** statement in a **SUB** or **FUNCTION** procedure must point to labels at module level. If no line label is given, execution always starts with the first executable line of the main module.

Program lines can have line numbers or alphanumeric labels, such as `OpenWindow:`. If an alphanumeric label is the target of a **RUN** statement, the compiler generates the error message `String expression required.`

You do not need to specify the filename extension in *filespec$*. The .BAS extension is assumed in the QBX environment, whereas the .EXE extension is assumed for compiled, stand-alone programs. If the program you wish to run has a different extension, you must give the extension. If the program name has no extension, the filename given must end with a period. For example, the following statement would execute `CATCHALL.EXE` from a BC-compiled program, and `CATCHALL.BAS` from within the QBX environment:

```
RUN "CATCHALL"
```

Programs running within the QBX environment must call only BASIC program files. The file is loaded and run as if it were a BASIC program; if it is not in the BASIC program format, execution halts. The error message that results varies, depending on the file's contents. Likewise, programs compiled with the compiler must not invoke BASIC source files, as these run only in the QBX environment.

An executable file need not have been written in BASIC. Any executable file can be run.

When running a program in the QBX environment, if an executable file matching the filename in the command line cannot be found, BASIC generates the error message `File not found` and control returns to BASIC. When running a program compiled by BC, BASIC generates the error message `File not found in module` *`module-name`* `at address` *`segment`*`:`*`offset`* and control returns to the operating system.

When the invoked program completes execution, control does not return to the invoking program. If the invoking program ran outside QBX, control returns to the operating system. If the invoking program ran under QBX, control returns to BASIC.

RUN closes all files and clears program memory before loading the designated program. The compiler does not support the R option from BASICA. (The R option keeps all open data files open.) If you want to run a different program, but leave open files open, use the **CHAIN** statement.

See Also

CHAIN

Example

This example shows how **RUN** *linenumber* resets all numeric variables to 0. As the line number following **RUN** increases in lines 60, 70, 80, and 90, the variables in the earlier statements lose their assigned values.

```
10 A = 9
20 B = 7
30 C = 5
40 D = 4
50 PRINT A, B, C, D
60 IF A = 0 THEN 70 ELSE RUN 20
70 IF B = 0 THEN 80 ELSE RUN 30
80 IF C = 0 THEN 90 ELSE RUN 40
90 IF D = 0 THEN END ELSE RUN 50
```

Output

```
9     7     5     4
0     7     5     4
0     0     5     4
0     0     0     4
0     0     0     0
```

SADD Function

Action Returns the address of a specified string variable.

Syntax SADD(*stringvariable$*)

Remarks The **SADD** function returns the address of a string as an offset (near pointer) from the current data segment. The offset is a two-byte integer. **SADD** is most often used in mixed-language programming to obtain far addresses before passing far strings to procedures written in other languages.

The argument *stringvariable$* is the string whose offset you want to determine. It can be a simple string variable or a single element of a string array. You cannot use fixed-length string arguments.

SADD can be used with both near and far strings. To obtain the segment address of a far string, use the **SSEG** function. In previous versions of BASIC, **SADD** was used only for near strings.

Note Do not add characters to the beginning or end of a string passed using **SADD** and **LEN**. Adding characters can cause BASIC to generate a run-time error. Use this function with caution, because strings can move in the BASIC string space (storage area) at any time.

See Also **BLOAD; BSAVE; DEF SEG; FRE; PEEK; POKE; SSEG; SSEGADD; VARPTR, VARSEG; VARPTR$**

Example The following example illustrates the use of the **SADD** and **SSEG** functions. **SADD** returns the offset address of a variable-length string. **SSEG** returns the segment of a variable-length string. Typically these functions are used in mixed-language programs or with **PEEK, POKE, BLOAD**, or **BSAVE**.

In this example, a string is created and then its offset and segment are calculated with **SADD** and **SSEG**. The information is then passed to a BASIC **SUB** procedure that mimics the performance of a non-BASIC print routine.

```
DEFINT A-Z
' Create the string.
Text$ = ".... a few well-chosen words"

' Calculate the offset, segment, and length of the string.
Offset = SADD(Text$)
Segment = SSEG(Text$)
Length = LEN(Text$)
```

```
' Pass these arguments to the print routine.
CALL printit(Segment, Offset, Length)

SUB printit (Segment, Offset, Length)
    CLS
    ' Set the segment for the PEEK function.
    DEF SEG = Segment
    FOR i = 0 TO Length - 1
        ' Get each character from memory, convert to ASCII, and display.
        PRINT CHR$(PEEK(i + Offset));
    NEXT i
END SUB
```

SAVEPOINT Function

Action Marks the beginning of a subset of ISAM database operations in a transaction.

Syntax **SAVEPOINT**

Remarks The **SAVEPOINT** function marks the beginning of a subset of operations within a transaction that can be rescinded using a **ROLLBACK** statement. The function returns an integer that refers to the savepoint.

Transactions are a way to group a series of ISAM operations so that you can commit them as a whole, rescind them all, or rescind operations since a designated savepoint. Use **BEGINTRANS** to indicate the beginning of a transaction and **COMMITTRANS** to commit all operations since the beginning of a transaction.

Use **ROLLBACK** with the argument *savepoint%* to return the data affected by a transaction to its state at that savepoint. Use **ROLLBACK** with no argument to return the data affected by the transaction to its state at the most recent savepoint, or at the beginning of the transaction if there are no intermediate savepoints.

ROLLBACK ALL rescinds all ISAM operations in a transaction and returns the data to its initial state at the beginning of the transaction.

If there is no transaction pending when you use **SAVEPOINT**, BASIC generates the error message `Illegal function call.`

See Also **BEGINTRANS, COMMITTRANS, ROLLBACK**

Example See the **BEGINTRANS** statement programming example, which uses the **SAVEPOINT** function.

SCREEN Function

Action Reads a specified character's ASCII value or its color from a specified screen location.

Syntax **SCREEN**(*line%*,*column%* ⟦,*colorflag%*⟧)

Remarks The **SCREEN** function uses the following arguments:

Argument	Description
line%	The line number of the character location on the screen. The argument *line%* is an integer expression whose valid range depends on the number of lines on screen.
column%	The column number of the character location on the screen. The argument *column%* is an integer expression with a value between 1 and 80, inclusive.
colorflag%	A numeric expression that determines which information is returned. When *colorflag%* is nonzero, **SCREEN** returns the number of the color attribute at the screen location. If *colorflag%* is 0 or is absent, the **SCREEN** function returns the ASCII character code.

Note In graphics screen modes, if the pattern on the screen at the given character location does not exactly match any pattern in the current ASCII character set, the **SCREEN** function will return the ASCII code for a space. If *colorflag%* is nonzero, the **SCREEN** function will return 0.

See Also Appendix A, "Keyboard Scan Codes and ASCII Character Codes," **COLOR**, **PALETTE**, **SCREEN** Statement

Examples If the character at (10,10) is `A`, then the following statement would return 65, the ASCII code for A, to the variable `X`:

```
X = SCREEN(10,10)
```

The following statement returns the color attribute of the character in the upper-left corner of the screen:

```
X = SCREEN(1, 1, 1)
```

SCREEN Statement

Action Sets the specifications for the graphics adapter and monitor.

Syntax **SCREEN** *mode%* [[, [[*colorswitch%*]] [[, [[*activepage%*]] [[,*visiblepage%*]]]]]]

Remarks The **SCREEN** statement sets a screen mode for a particular combination of display and adapter. Later sections in this entry describe the available modes for specific adapters. The **SCREEN** statement uses the following arguments:

Argument	Description
mode%	An integer constant or expression that selects a screen mode for a particular combination of display and adapter (see "Summary of Screen Modes" later in this entry for information on what the acronyms represent). There also are more details on *mode%* later in this entry.
colorswitch%	Integer expression that switches composite monitor display between color and monocolor (modes 0 and 1 only). See below for more details on *colorswitch%*.
activepage%	Integer expression that identifies the screen page that text or graphics output is written to.
visiblepage%	Integer expression that identifies the screen page that is displayed.

The *colorswitch%* is effective only for screen modes 0 and 1 and for composite monitors. In screen mode 0, use a *colorswitch%* value of 0 to disable color, and a non-zero value to enable color. In screen mode 1, use a non-zero value to disable color and 0 to enable color.

See "Adapters, Screen Modes, and Displays" later in this entry for the valid ranges for *activepage%* and *visiblepage%* for each graphics adapter. The "Attributes and Colors" section lists the default color attributes for different screen modes. If *mode%* is an expression, rather than a constant, and if you know your program will not use certain screen modes, you can achieve a smaller .EXE file by linking one or more stub files:

If not using screen modes:	Link to file:
1 or 2	NOCGA.OBJ
3	NOHERC.OBJ
4	NOOGA.OBJ
7, 8, 9, or 10	NOEGA.OBJ
11, 12, or 13	NOVGA.OBJ

If you do not need graphics in your custom run-time module (because the programs use screen mode 0 only), you can save 15K by creating the run-time module with NOGRAPH.OBJ.

Summary of Screen Modes

The following lists summarize each of the screen modes. The color adapters referred to are the IBM Monochrome Display and Printer Adapter (MDPA), the IBM Color Graphics Adapter (CGA), the IBM Enhanced Graphics Adapter (EGA), the IBM Video Graphics Array (VGA), the IBM Multicolor Graphics Array (MCGA), and the Olivetti Color Adapter. The Hercules Graphics Card, Graphics Card Plus and InColor adapters are supported, but only with monochrome monitors.

Note Many screen modes support more than one combination of rows and columns. See the **WIDTH** statement for more information about changing the number of rows and columns on the display.

`SCREEN 0`

- MDPA, CGA, EGA, MCGA, Hercules, Olivetti, or VGA Adapter Boards
- Text mode only
- 40 x 25, 40 x 43, 40 x 50, 80 x 25, 80 x 43, or 80 x 50 text format with 8 x 8 character box size (8 x 14, 9 x 14, or 9 x 16 with EGA or VGA)
- 16 colors assigned to two color attributes
- 16 colors assigned to any of 16 color attributes with CGA or EGA
- 64 colors assigned to any of 16 color attributes with EGA or VGA

`SCREEN 1`

- CGA, EGA, VGA, or MCGA Adapter Boards
- 320 x 200 graphics
- 40 x 25 text format, 8 x 8 character box
- 16 background colors and one of two sets of three foreground colors assigned using **COLOR** statement with CGA
- 16 colors assigned to four color attributes with EGA, VGA or MCGA

`SCREEN 2`

- CGA, EGA, VGA, or MCGA Adapter Boards
- 640 x 200 graphics
- 80 x 25 text format with character box size of 8 x 8
- 2 colors (black and white) with CGA
- 16 colors assigned to two color attributes with EGA or VGA

The following lists summarize screen modes used with other adapter boards.

`SCREEN 3`

- Hercules, Olivetti, or AT&T Adapter Boards
- Hercules adapter required, monochrome monitor only
- 720 x 348 graphics
- 80 x 25 text format, 9 x 14 character box
- Two screen pages (one only if a second display adapter is installed)
- **PALETTE** statement not supported

`SCREEN 4`

- Hercules, Olivetti, or AT&T Adapter Boards
- 640 x 400 graphics
- 80 x 25 text format, 8 x 16 character box
- One of 16 colors assigned as the foreground color (selected by the **COLOR** statement); background fixed at black
- Supports Olivetti Personal Computers models M24, M240, M28, M280, M380, M380/C, and M380/T, and AT&T Personal Computers 6300 series

Warning Olivetti personal computers running 3XBOX under OS/2 should avoid screen mode 4.

`SCREEN 7`

- EGA or VGA Adapters
- 320 x 200 graphics
- 40 x 25 text format, character box size 8 x 8
- 32K page size, pages 0–1 (64K adapter memory), 0–3 (128K), or 0–7 (256K)
- Assignment of 16 colors to any of 16 color attributes

`SCREEN 8`

- EGA or VGA Adapters
- 640 x 200 graphics
- 80 x 25 text format, 8 x 8 character box
- 64K page size, pages 0 (64K adapter memory), 0–1 (128K), or 0–3 (256K)
- Assignment of 16 colors to any of 16 color attributes

`SCREEN 9`

- EGA or VGA Adapters
- 640 x 350 graphics
- 80 x 25 or 80 x 43 text format, 8 x 14 or 8 x 8 character box size
- 64K page size, page 0 (64K adapter memory); 128K page size, pages 0 (128K adapter memory) or 0–1 (256K)
- 16 colors assigned to four color attributes (64K adapter memory), or 64 colors assigned to 16 color attributes (more than 64K adapter memory)

`SCREEN 10`

- EGA or VGA adapters, monochrome monitor only
- 640 x 350 graphics, monochrome monitor only
- 80 x 25 or 80 x 43 text format, 8 x 14 or 8 x 8 character box size
- 128K page size, pages 0 (128K adapter memory) or 0–1 (256K)
- Up to nine shades of gray assigned to four color attributes

`SCREEN 11`

- VGA or MCGA adapters
- 640 x 480 graphics
- 80 x 30 or 80 x 60 text format, character box size of 8 x 16 or 8 x 8
- Assignment of up to 256K colors to two color attributes

`SCREEN 12`

- VGA adapter
- 640 x 480 graphics
- 80 x 30 or 80 x 60 text format, character box size of 8 x 16 or 8 x 8
- Assignment of up to 256K colors to 16 color attributes

`SCREEN 13`

- VGA or MCGA adapters
- 320 x 200 graphics
- 40 x 25 text format, character box size of 8 x 8
- Assignment of up to 256K colors to up to 256 color attributes

Screen Modes, Adapters, and Displays

This section describes the screen modes available for each adapter. If the display device also is a factor in choosing a screen mode, it is listed. The IBM Monochrome Display and Printer Adapter (MDPA) must be used with a monochrome display. Only `SCREEN 0`, text mode, can be used with the MDPA.

Table 1.10 describes the screen mode available with the MDPA.

Table 1.10 MDPA Screen Modes

Mode	Rows and columns	Color attributes	Display colors	Resolution	Pages
0	80x25	16	3	720x350	1

Table 1.11 summarizes the screen modes available with Hercules Adapters.

Table 1.11 Hercules Screen Modes

Mode	Rows and columns	Color attributes	Display colors	Resolution	Pages
0	80x25	16	3	720x348	1
3	80x25	16	1	720x348	2

The IBM Color Graphics Adapter (CGA) and Color Display typically are paired. This combination permits running text-mode programs, and both medium-resolution and high-resolution graphics programs.

Table 1.12 summarizes the screen modes available with the CGA.

Table 1.12 CGA Screen Modes

Mode	Rows and columns	Color attributes	Resolution	Pages
0	40x25	16	320x200	8
	80x25	16	640x200	4
1	40x25	4	320x200	1
2	80x25	2	640x200	1

The IBM Enhanced Graphics Adapter (EGA) can be used with either the IBM Color Display or the Enhanced Color Display. In modes 0, 1, 2, 7, and 8, these pairings produce similar results, except for the following differences:

- Border color can't be set on an Enhanced Color Display when it is in 640 x 350 text mode.
- The text quality is better on the Enhanced Color Display (an 8 x 14 character box for Enhanced Color Display versus an 8 x 8 character box for Color Display).

Screen mode 9 takes full advantage of the capabilities of the Enhanced Color Display. Mode 9 provides for the highest resolution possible for the EGA/Enhanced Color Display configuration. Programs written for this mode will not work for any other hardware configuration except the VGA.

Table 1.13 summarizes the screen modes that can be used with an EGA.

Table 1.13 EGA Screen Modes

Mode	Rows & clmns	Display[1]	Color Attributes	Display Colors	Page Resolution	Size	Pages
0	40x25	C	16	16	320x200	N/A	8
	40x25	E	16	64	320x350	N/A	8
	40x43	E	16	64	320x350	N/A	8[2]
	80x25	C	16	16	640x200	N/A	8[2]
	80x25	E	16	64	640x350	N/A	8[2]
	80x25	C	16	16	640x200	N/A	8[2]
	80x25	M	16	3	720x350	N/A	8[2]
	80x43	E	16	64	640x350	N/A	4[2]
	80x43	M	16	3	720x350	N/A	4[2]
1	40x25	N/A	4	16	320x200	16K	1
2	80x25	N/A	2	16	640x200	16K	1
7	40x25	N/A		16	320x200	32K	See note[3]
8	80x25	N/A	16	16	640x200	64K	See note[3]
9[4]	80x25	E	4	64	640x350	64K	1
	80x43	E	4	64	640x350	64K	1
	80x25	E	16	64	640x350	128K	See note[3]
	80x43	E	16	64	640x350	128K	See note[3]
10	80x25	M	4	9	640x350	64K	See note[5]
	80x43	M	4	9	640x350	64K	See note[5]

1 C = Color display, E = Enhanced color display, M = Monochrome display, N/A = Not applicable (either color display or enhanced color display).
2 Pages = Screen memory divided by page size. Eight page maximum, one page minimum.
3 Pages = Screen memory divided by 2 divided by page size. Eight page maximum, one page minimum.
4 Number of pages is halved with 64K.
5 The first two entries under mode 9 are for an EGA with 64K of screen memory. The next two entries assume more than 64K of screen memory.

Only the EGA and VGA can be paired with the IBM Monochrome display to run programs in screen mode 10. This mode can be used to display monochrome graphics at a very high resolution with the optional effects of blinking and high intensity.

The following two tables summarize the color attributes, display colors, and effects for screen mode 10 used with a monochrome display.

Tables 1.14 and 1.15 summarize EGA and VGA adapters used with monochrome display (`SCREEN 10`).

Table 1.14 Color Attributes: SCREEN 10, Monochrome Display

Color attribute	Default effect
0	Off
1	On, normal intensity
2	Blink
3	On, high intensity

Table 1.15 Display Color Values: SCREEN 10, Monochrome Display

Display color	Effect
0	Off
1	Blink, off to on
2	Blink, off to high intensity
3	Blink, on to off
4	On
5	Blink, on to high intensity
6	Blink, high intensity to off
7	Blink, high intensity to on
8	High intensity

The IBM Video Graphics Array (VGA) adapter offers significantly enhanced text and graphics in all modes. Table 1.16 summarizes the modes available with the VGA.

Table 1.16 VGA Screen Modes

Mode	Rows & columns	Color attributes	Display colors	Resolution	Page size	Pages
0	40x25	16	64	360x400		8
	40x43	16	64	320x350		8
	40x50	16	64	320x400		4
	80x25	16	64	720x400		8
	80x43	16	64	640x350		4
	80x43	16	3	720x350		4
	80x50	16	4	640x400		4
	80x50	16	3	720x400		4
1	40x25	4	16	320x200	16K	1
2	80x25	2	16	640x200	16K	1
7	40x25	16	16	320x200	32K	See note[1]
8	80x25	16	16	640x200	64K	See note[1]
9	80x25	16	64	640x350	128K	See note[1]
	80x43	16	64	640x350	128K	See note[1]
10	80x25	4	9	640x350	64K	See note[2]
	80x43	4	9	640x350	64K	See note[2]
11	80x30	2	256	640x480	64K	1
	80x60	2	256	640x480	64K	1
12	80x30	16	256	640x480	256K	1
	80x60	16	256	640x480	256K	1
13	40x25	256	256	320x200	64K	1

1 Pages = Screen memory divided by page size. Eight page maximum.
2 Pages = Screen memory divided by 2 divided by page size. Eight page maximum.

See the **PALETTE** statement for a description of how the VGA calculates color values.

The IBM Multicolor Graphics Array (MCGA) combines the modes of the CGA with the very high resolution and 256K color modes of the VGA to provide enhanced text and graphics in all modes. Table 1.17 summarizes the modes supported by the MCGA.

Table 1.17 MCGA Screen Modes

Mode	Rows & columns	Color attributes	Display colors	Resolution	Page size	Pages
0	40x25	16		320x400		8
	80x25	16		640x400		8
1	40x25	4		320x200	16K	1
2	80x25	2		640x200	16K	1
11	80x30	2	256K	640x480	64K	1
	80x60	2	256K	640x480	64K	1
13	40x25	256	256K	320x200	64K	1

The MCGA uses the same display color values as the VGA. For a description of how the MCGA calculates display color values, see the **PALETTE** statement.

Attributes and Colors

For various screen modes and display hardware configurations, different color-attribute and display-color settings exist. (See the **PALETTE** statement for a discussion of color attributes and display colors.) The majority of these color-attribute and display-color configurations are summarized in Tables 1.18–1.20.

Table 1.18 Color Attributes and Default Display Colors for Screen Modes 0, 7, 8, 9 [1] 12, and 13

	Color monitor		Monochrome monitor	
Color attribute	Default display color value[2]	Displayed color	Default display color value[3]	Displayed color
0	0	Black	0	Off
1	1	Blue		Underlined[4]
2	2	Green	1	On[4]
3	3	Cyan	1	On[4]
4	4	Red	1	On[4]
5	5	Magenta	1	On[4]
6	6	Brown	1	On[4]
7	7	White	1	On[4]
8	8	Gray	0	Off
9	9	Light Blue		High-intensity underlined
10	10	Light green	2	High intensity
11	11	Light cyan	2	High intensity
12	12	Light red	2	High intensity
13	13	Light magenta	2	High intensity
14	14	Yellow	2	High intensity
15	15	High intensity	0	Off-white

1 For VGA. Also for EGA with video memory > 64K.
2 EGA display color values. VGA and MCGA use display-color values that produce visually equivalent colors.
3 Only for mode 0 monochrome.
4 Off when used for background.

Table 1.19 Color Attributes and Default Display Colors for Screen Modes 1 and 9 [1]

	Color monitor		Monochrome monitor	
Color attribute	Default display color value[2]	Displayed color	Default display color value[3]	Displayed color
0	0	Black	0	Off
1	11	Light cyan	2	High intensity
2	13	Light magenta	2	High intensity
3	15	High-intensity	0	Off white

1 EGA with video memory <= 64K.
2 EGA display color values. VGA and MCGA use display-color values that produce visually equivalent colors.
3 Only for mode 0 monochrome.

Table 1.20 Color Attributes and Default Display Colors for Screen Modes 2 and 11

	Color monitor		Monochrome monitor	
Color attribute	Default display color value[1]	Displayed color	Default display color value[2]	Displayed color
0	0	Black	0	Off
1	15	High intensity	0	Off white

[1] EGA display color values. VGA and MCGA use display color values that produce visually equivalent colors.
[2] Only for mode 0 monochrome.

Note

In OS/2 protected mode, BASIC supports only screen modes 0–2 and does not support the *activepage%* and *visiblepage%* arguments.

See Also

COLOR, **PALETTE**, **SCREEN** Function, **WIDTH**

Example

The following example sets up a multipage EGA or VGA mode 7 (320x200) display using the **SCREEN** statement. A help screen is displayed for several seconds then copied (using the **PCOPY** statement) to page 2 for later use. A cube is displayed and rotated on the screen. By pressing Shift+F1, the user can again see the help screen, after which the cube display is resumed.

```
DEFINT A-Z
' Define a macro string to draw a cube and paint its sides.
One$ = "BR30 BU25 C1 R54 U45 L54 D45 BE20 P1, 1G20 C2 G20"
Two$ = "R54 E20 L54 BD5 P2, 2 U5 C4 G20 U45 E20 D45 BL5 P4, 4"
Plot$ = One$ + Two$
' Initialize values for active page, visual page, and help page.
APage% = 0 : VPage% = 1 : HPage% = 2
Angle% = 0
' Create a HELP screen on the visual page.
SCREEN 7, 0, VPage%, VPage%
LOCATE 1, 18
PRINT "HELP"
LOCATE 5, 1
PRINT "Press 'Alt+F1' keys to see this HELP"
PRINT "screen while the cube is rotating"
PRINT "around the center of the screen."
PRINT
PRINT "After a brief delay, the cube will"
PRINT "resume at the next point in its rotation."
PRINT
PRINT "Press any other key to exit the program."
```

```
' Put a copy of the help screen in page 2.
PCOPY VPage%, HPage%
SLEEP 5
DO
    SCREEN 7, 0, APage%, VPage%
    ' Clear the active page.
    CLS 1
    ' Rotate the cube Angle% degrees.
    DRAW "TA" + STR$(Angle%) + Plot$
    ' Angle% is some multiple of 15 degrees.
    Angle% = (Angle% + 15) MOD 360
    ' Drawing is complete. Make the cube visible in its
    '  new position by swapping the active and visual pages.
    SWAP APage%, VPage%
    ' Check the keyboard input.
    Kbd$ = INKEY$
    SELECT CASE Kbd$
        ' If Shift+F1 is pressed, show the HELP page.
        CASE CHR$(0) + CHR$(104)
            PCOPY HPage%, APage%
            SLEEP 3
        ' Do nothing if no key is pressed.
        CASE ""
        CASE ELSE
            EXIT DO
    END SELECT
LOOP
END
```

SEEK Function

Action Returns the current file position.

Syntax **SEEK**(*filenumber%*)

Remarks The argument *filenumber%* is the number used in the **OPEN** statement to open the file. **SEEK** returns a value between 1 and 2,147,483,647, inclusive (equivalent to $2^{31}-1$).

SEEK returns the number of the next record read or written when used on random-access files. For files opened in binary, output, append, or input mode, **SEEK** returns the byte position in the file where the next operation is to take place. The first byte in a file is 1.

SEEK returns zero when used on an ISAM table or on BASIC devices (SCRN, CONS, KYBD, COM*n*, LPT*n*, PIPE) that do not support **SEEK**.

See Also **GET** (File I/O); **OPEN**; **PUT** (File I/O); **SEEK** Statement; **SEEKGT**, **SEEKGE**, **SEEKEQ** Statements

Example See the **SEEK** statement programming example, which uses the **SEEK** function.

SEEK Statement

Action Sets the position in a file for the next read or write.

Syntax **SEEK** [[#]]*filenumber%*,*position&*

Remarks The **SEEK** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
position&	A numeric expression that indicates where the next read or write occurs. The *position&* value must be between 1 and 2,147,483,647 (equivalent to 2^{31} –1), inclusive.

For files opened in random-access mode, *position&* is the number of a record in the file.

For files opened in binary, input, output, or append mode, *position&* is the byte position relative to the beginning of the file. The first byte in a file is at position 1; the second byte is at position 2, and so on. After a **SEEK** operation, the next file I/O operation starts at the specified byte position.

Note Record numbers on a **GET** or **PUT** override the file positioning done by **SEEK**.

Performing a file write after doing a **SEEK** operation beyond the end of a file extends the file. If you attempt a **SEEK** operation to a negative or zero position, BASIC generates the error message `Bad record number.`

BASIC leaves the file position unchanged when you use **SEEK** on an ISAM table or on BASIC devices (SCRN, CONS, KYBD, COM*n*, LPT*n*, PIPE) that do not support **SEEK**.

See Also **GET** (File I/O); **OPEN**; **PUT** (File I/O); **SEEK** Function; **SEEKGT**, **SEEKGE**, **SEEKEQ** Statements

Example The following example uses a combination of the **SEEK** function and **SEEK** statement to move the file position exactly one record back and rewrite the record if a variable is true (nonzero):

```
CONST FALSE = 0, TRUE = NOT FALSE
' Define record fields and a user-type variable.
TYPE TestRecord
   NameField  AS STRING * 20
   ScoreField AS SINGLE
END TYPE
DIM RecordVar AS TestRecord
DIM I AS LONG
```

```
' This part of the program creates the random-access file used by the
' second part of the program, which demonstrates the SEEK statement.
OPEN "TESTDAT2.DAT" FOR RANDOM AS #1 LEN = LEN(RecordVar)
RESTORE
READ NameField$, ScoreField
I = 0
DO WHILE NameField$ <> "END"
   I = I + 1
   RecordVar.NameField = NameField$
   RecordVar.ScoreField = ScoreField
   PUT #1, I, RecordVar
   READ NameField$, ScoreField
LOOP
CLOSE #1
  DATA "John Simmons", 100
  DATA "Allie Simpson", 95
  DATA "Tom Tucker", 72
  DATA "Walt Wagner", 90
  DATA "Mel Zucker", 92
  DATA "END", 0

' Open the test data file.
DIM FileBuffer AS TestRecord
OPEN "TESTDAT2.DAT" FOR RANDOM AS #1 LEN = LEN(FileBuffer)
' Calculate number of records in the file.
Max = LOF(1) \ LEN(FileBuffer)
' Read and print contents of each record.
FOR I = 1 TO Max
   GET #1, I, FileBuffer
   IF FileBuffer.NameField = "Tom Tucker" THEN
      ReWriteFlag = TRUE
      EXIT FOR
   END IF
NEXT I
IF ReWriteFlag = TRUE THEN
   ' Back up file by the length of the record variable that
   ' is used to write to the file.
   FileBuffer.ScoreField = 100
   SEEK #1, SEEK(1) - LEN(RecordVar)
   PUT #1, , RecordVar
END IF
CLOSE #1
KILL "TESTDAT2.DAT"
END
```

SEEKGT, SEEKGE, SEEKEQ Statements

Action Causes the first matching record in an ISAM table to become the current record.

Syntax **SEEKGT** [[#]] *filenumber%* [[,*keyvalue*[[,*keyvalue*]]...]]
SEEKGE [[#]] *filenumber%* [[,*keyvalue*[[,*keyvalue*]]...]]
SEEKEQ [[#]] *filenumber%* [[,*keyvalue*[[,*keyvalue*]]...]]

Remarks The **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements cause the first matching record in the table, according to the current index, to become the current record. The following table shows which value of the current index column causes the record to become current for each statement.

Statement	Current-index column value
SEEKGT	Greater than *keyvalue*
SEEKGE	Greater than, or equal to, *keyvalue*
SEEKEQ	Equal to *keyvalue*

The argument *filenumber%* is the number used in the **OPEN** statement to open the table. The argument *keyvalue* is an expression fewer than 256 characters long that is evaluated based on the operand condition in the **SEEK***operand* keyword. If more than one *keyvalue* argument is specified, BASIC assumes the current index is based on the combination of their values.

If no match is found, the current index position is at the end of the table and there is no current record.

If the number of *keyvalues* is greater than the number of values that makes up the current index, BASIC generates the error message `Syntax Error.`

No error is generated if the number of *keyvalues* is less than the number of values that makes up the current index. The seek either fails or is based on the *keyvalues* supplied:

- **SEEKEQ** with too few *keyvalues* always fails.
- **SEEKGE** with too few *keyvalues* is equivalent to a **SEEKGE** with the same arguments.
- **SEEKGT** with too few *keyvalues* seeks for the first record that matches the *keyvalues* supplied.

BASIC removes trailing spaces from strings used in a seek.

Note The *keyvalue* expression must be explicitly typed if it is a DOUBLE or CURRENCY value.

See Also **MOVEFIRST**, **MOVELAST**, **MOVENEXT**, **MOVEPREVIOUS** Statements; **OPEN**

Example The following example uses the **CREATETABLE** statement to create a new table in an ISAM file and uses the **SEEKGE**, **RETRIEVE**, **UPDATE**, and **INSERT** statements to insert records into the file. It uses the **LOF** function to display the number of records in the new table and then destroys the table with **DELETETABLE**.

The program uses a file called BOOKS.MDB, the sample ISAM file that SETUP copies to your disk.

```
DEFINT A-Z
TYPE BookRec
    IDNum AS DOUBLE              ' Unique ID number for each book.
    Title AS STRING * 50         ' Book's title.
    Publisher AS STRING * 50     ' Book's publisher.
    Author AS STRING * 36        ' Book's author.
    Price AS CURRENCY            ' Book's price.
END TYPE

CONST Database = "BOOKS.MDB"     ' Name of the disk file.
DIM Library AS BookRec           ' Variable for current record.
DIM MinPrice AS CURRENCY         ' SEEK criteria.

CLS
DO
INPUT "Display books that cost as much or more than "; MinPrice
    IF MinPrice < 0 THEN PRINT "Positive values only, please."
LOOP UNTIL MinPrice > 0

' Open existing table.
LibraryFile = FREEFILE
OPEN Database FOR ISAM BookRec "BooksStock" AS LibraryFile
CREATEINDEX LibraryFile, "Library", 0, "Price"
SETINDEX LibraryFile, "Library"

' Create and open a new table.
NewFile = FREEFILE
OPEN Database FOR ISAM BookRec "PricyBooks" AS NewFile

' Fill new table with records for all books with price >= MinPrice.
SEEKGE LibraryFile, MinPrice
DO
    RETRIEVE LibraryFile, Library
    INSERT NewFile, Library
    MOVENEXT LibraryFile
LOOP UNTIL EOF(LibraryFile)
```

```
' First time through loop get price increase;
' second time display new price.
FOR count = 1 TO 2
    CLS
    PRINT SPC(18); LOF(NewFile); "books cost at least ";
    PRINT USING ("$###.##"); MinPrice
    PRINT " ID Number"; SPC(3); "Title"; SPC(20); "Author";
    PRINT SPC(11); "Publisher"; SPC(10); "Price"
    VIEW PRINT 3 TO 20
    MOVEFIRST NewFile
    DO
        RETRIEVE NewFile, Library
        PRINT Library.IDNum; " "; LEFT$(Library.Title, 20);
        IF LEN(RTRIM$(Library.Title)) > 20 THEN
            PRINT "...  ";
        ELSE
            PRINT "     ";
        END IF
        PRINT LEFT$(Library.Author, 15); "  ";
        PRINT LEFT$(Library.Publisher, 16); "  ";
        PRINT USING ("$###.##"); Library.Price
        MOVENEXT NewFile
    LOOP UNTIL EOF(NewFile)
    IF count = 1 THEN
        VIEW PRINT 20 TO 24: LOCATE 20, 1
        DO
            INPUT "Increase cost by how much (0-100%)"; increase
            IF increase < 0 OR increase > 100 THEN PRINT "Illegal value"
        LOOP UNTIL increase >= 0 AND increase <= 100
        ' Update records in PricyBooks.
        MOVEFIRST NewFile
        DO
            RETRIEVE NewFile, Library
            Library.Price = (Library.Price * (100 + increase)) / 100
            ' Overwrite record with increased price.
            UPDATE NewFile, Library
            CHECKPOINT       ' Force ISAM to flush the buffer to disk.
            MOVENEXT NewFile
        LOOP UNTIL EOF(NewFile)
    END IF
    VIEW PRINT 1 TO 19
NEXT count
' Destroy index and temporary table, close files.
DELETEINDEX LibraryFile, "Library"
CLOSE
DELETETABLE Database, "PricyBooks"
```

SELECT CASE Statement

Action Executes one of several statement blocks depending on the value of an expression.

Syntax

SELECT CASE *testexpression*
CASE *expressionlist1*
  ⟦*statementblock-1* ⟧
⟦**CASE** *expressionlist2*
  ⟦*statementblock-2*⟧ ⟧
⟦**CASE ELSE**
  ⟦*statementblock-n*⟧ ⟧
END SELECT

Remarks The following list describes the parts of the **SELECT CASE** statement:

Argument	Description
testexpression	Any numeric or string expression.
statementblock-1, *statementblock-2*, *statementblock-n*	The elements *statementblock-1* to *statementblock-n* consist of any number of statements on one or more lines.
expressionlist1, *expressionlist2*	These elements can have any of the three following forms: *expression* ⟦,*expression*⟧... *expression* **TO** *expression* **IS** *relational-operator expression*

The following list describes the parts of *expressionlist*:

Argument	Description
expression	Any numeric or string expression. The type of the expression must be compatible with the type of *testexpression*. (The type of the expression will be coerced to the same type as *testexpression*. For example, if *testexpression* is an integer, *expressionlist* can contain a double-precision data type.)

relational-operator

Any of the following operators:

Symbol	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal
=	Equal

If *testexpression* matches the expression list associated with a **CASE** clause, then the statement block following that **CASE** clause is executed up to the next **CASE** clause or, for the last one, up to **END SELECT**. Control then passes to the statement following **END SELECT**.

If you use the **TO** keyword to indicate a range of values, the smaller value must appear first. For example, the statements associated with the line `CASE -1 TO -5` are not executed if *testexpression* is `-4`. The line should be written as `CASE -5 TO -1`.

You can use a relational operator only if the **IS** keyword appears. If **CASE ELSE** is used, its associated statements are executed only if *testexpression* does not match any of the other **CASE** selections. It is a good idea to have a **CASE ELSE** statement in your **SELECT CASE** block to handle unforeseen *testexpression* values.

When there is no **CASE ELSE** statement and no expression listed in the **CASE** clauses matches *testexpression*, program execution continues normally.

You can use multiple expressions or ranges in each **CASE** clause. For example, the following line is valid:

```
CASE 1 TO 4, 7 TO 9, 11, 13, IS > MaxNumber%
```

You also can specify ranges and multiple expressions for strings:

```
CASE "everything", "nuts" TO "soup", TestItem$
```

CASE matches strings that are exactly equal to `everything`, the current value of `TestItem$`, or that fall between `nuts` and `soup` in alphabetical order.

Strings are evaluated according to the ASCII values of their characters. Lowercase letters have larger ASCII values than uppercase letters, so this statement is true:

```
nuts > Nuts > NUTS
```

If an expression appears in more than one **CASE** clause, only the statements associated with the first appearance of the expression are executed.

SELECT CASE statements can be nested. Each **SELECT CASE** statement must have a matching **END SELECT** statement.

Examples

In the first example, **SELECT CASE** is used to take different actions based on the input value:

```
INPUT "Enter acceptable level of risk (1-10): ", Total
SELECT CASE Total
CASE IS >= 10
   PRINT "Maximum risk and potential return."
   PRINT "Choose stock investment plan."
CASE 6 TO 9
   PRINT "High risk and potential return."
   PRINT "Choose corporate bonds."
CASE 2 TO 5
   PRINT "Moderate risk and return."
   PRINT "Choose mutual fund."
CASE 1
   PRINT "No risk, low return."
   PRINT "Choose IRA."
CASE ELSE
   PRINT "Response out of range."
END SELECT
```

Output

```
Enter acceptable level of risk (1-10): 10
Maximum risk and potential return.
Choose stock investment plan.

Enter acceptable level of risk (1-10): 0
Response out of range.
```

In the following program, the **SELECT CASE** statement is used to take different actions based on the ASCII value of a character.

```
' Function and control key constants.
CONST ESC = 27, DOWN = 80, UP = 72, LEFT = 75, RIGHT = 77
CONST HOME = 71, ENDKEY = 79, PGDN = 81, PGUP = 73
CLS
```

```
PRINT "Press Escape key to end."
DO
   ' Get a function or ASCII key.
   DO
      Choice$ = INKEY$
   LOOP WHILE Choice$ = ""
   IF LEN(Choice$) = 1 THEN   ' Handle ASCII keys.
      SELECT CASE ASC(Choice$)
      CASE ESC
         PRINT "Escape key"
         END
      CASE IS < 32, 127
         PRINT "Control code"
      CASE 48 TO 57
         PRINT "Digit: "; Choice$
      CASE 65 TO 90
         PRINT "Uppercase letter: "; Choice$
      CASE 97 TO 122
         PRINT "Lowercase letter: "; Choice$
      CASE ELSE
         PRINT "Punctuation: "; Choice$
      END SELECT
   ELSE   ' Convert 2-byte extended code to 1-byte ASCII code.
      Choice$ = RIGHT$(Choice$, 1)
      SELECT CASE Choice$
      CASE CHR$(DOWN)
         PRINT "DOWN direction key"
      CASE CHR$(UP)
         PRINT "UP direction key"
      CASE CHR$(PGDN)
         PRINT "PGDN key"
      CASE CHR$(PGUP)
         PRINT "PGUP key"
      CASE CHR$(HOME)
         PRINT "HOME key"
      CASE CHR$(ENDKEY)
         PRINT "END key"
      CASE CHR$(RIGHT)
         PRINT "RIGHT direction key"
      CASE CHR$(LEFT)
         PRINT "LEFT direction key"
      CASE ELSE
         BEEP
      END SELECT
   END IF
LOOP
```

SETINDEX Statement

Action Makes the specified ISAM table index the current index.

Syntax **SETINDEX** [[#]]*filenumber%* [[, *indexname$*]]

Remarks The **SETINDEX** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the table.
indexname$	The name used in the **CREATEINDEX** statement to create the index. If no *indexname$* argument or set of double quotation marks ("") is specified, the NULL index becomes the current index. The NULL index represents the order in which records were added to the file.

After **SETINDEX** makes an index current, the current record is the first record according to that index.

See Also **CREATEINDEX, DELETEINDEX, GETINDEX$**

Example See the **CREATEINDEX** statement programming example, which uses the **SETINDEX** statement.

SETMEM Function

Action Changes the amount of memory used by the far heap—the area where far objects are stored—and returns information about the far heap.

Syntax **SETMEM**(*numeric-expression&*)

Remarks The **SETMEM** function increases or decreases the far heap by the number of bytes indicated by *numeric-expression&*. If *numeric-expression&* is negative, **SETMEM** decreases the far heap by the indicated number of bytes. If *numeric-expression&* is positive, **SETMEM** attempts to increase the far heap by the number of bytes. If **SETMEM** cannot change the far heap by the requested number of bytes, it reallocates as many bytes as possible.

SETMEM can be used in mixed-language programming to decrease the far heap so procedures in other languages can dynamically allocate far memory.

The information that the **SETMEM** function returns about the far heap depends on the value of *numeric-expression&*, as follows:

If *numeric-expression&* is:	SETMEM returns:
Positive or negative	Total number of bytes in the far heap.
0	Current size of far heap.

Note A first call to **SETMEM** trying to increase the far heap has no effect because BASIC allocates as much memory as possible to the far heap when a program starts.

When programming with OS/2 protected mode, the **SETMEM** function performs no other function than to return a dummy value. This value is the previous **SETMEM** value adjusted by the **SETMEM** argument. The initial value for **SETMEM** is 655,360.

Example

In the following example, **SETMEM** is used to free memory for a C routine that uses the C function malloc to get dynamic memory. The C routine must be separately compiled and then put in a Quick library or linked to the BASIC program.

```
DECLARE SUB CFunc CDECL (BYVAL X AS INTEGER)

' Decrease the size of the far heap so CFunc can use
' malloc to get dynamic memory.
BeforeCall = SETMEM(-2048)

CFunc(1024%)     ' Call the C routine.

' Return the memory to the far heap; use a larger value so
' all space goes back into the heap.
AfterCall = SETMEM(3500)
IF AfterCall <= BeforeCall THEN PRINT "Memory not reallocated."
END
```

This routine is compiled using the large memory model, so calls to the C function malloc use the far space freed by the BASIC program.

```
void far cfunc(bytes)
int bytes;
{
   char *malloc();
   char *workspace;

   /* Allocate working memory using amount BASIC freed. */
   workspace=malloc((unsigned) bytes);

   /* Working space would be used here. */

   /* Free memory before returning to BASIC. */
   free(workspace);
}
```

SetUEvent Routine

Action Sets the BASIC entry point that causes a user-defined event.

Syntax **SetUEvent**

Remarks **SetUEvent** is used in user-event trapping.

SetUEvent signals an event for the **ON UEVENT** event-handling routine.

The **SetUEvent** routine is a part of BASIC, and is automatically included in compiled applications or when running QBX with the /L command-line option. Your interrupt-service routine must call **SetUEvent**; it is the only way to alert your program that the event has occurred. You can call **SetUEvent** from any non-BASIC language.

To use the **SetUEvent** routine in the QBX environment, use any Quick library. Outside of the QBX environment, you do not have to link to another library.

See Also **ON** *event*, **UEVENT**

Example See the **UEVENT** statements programming example, which uses the **SetUEvent** routine.

SGN Function

Action Indicates the sign of a numeric expression.

Syntax SGN(*numeric-expression*)

Remarks The **SGN** function returns a value depending on the sign of its argument:

- If *numeric-expression* > 0, then **SGN**(*numeric-expression*) returns 1.
- If *numeric-expression* = 0, then **SGN**(*numeric-expression*) returns 0.
- If *numeric-expression* < 0, then **SGN**(*numeric-expression*) returns –1.

Example The following example calculates and prints the solution for the input quadratic (or second-degree) equation. The program uses the sign of a test expression to determine how to calculate the solution.

```
CONST NoRealSoln=-1, OneSoln=0, TwoSolns=1
' Input coefficients of quadratic equation:
' ax^2 + bx + c = 0.
INPUT;"a = ",   A
INPUT;",  b = ",B
INPUT ",  c = ",C
Test = B^2 - 4 * A * C
SELECT CASE SGN(Test)
   CASE  NoRealSoln
      PRINT "This equation has no real-number solutions."
   CASE  OneSoln
      PRINT "This equation has one solution: ";
      PRINT -B/(2 * A)
   CASE  TwoSolns
      PRINT "This equation has two solutions: ";
      PRINT (-B + SQR(Test))/(2 * A) " and ";
      PRINT (-B - SQR(Test))/(2 * A)
END SELECT
```

Output

```
a = 3,  b = -4,  c = 1
This equation has two solutions:  1  and  .3333333
```

SHARED Statement

Action Gives a **SUB** or **FUNCTION** procedure access to variables declared at the module level without passing them as parameters.

Syntax **SHARED** *variable*⟦( )⟧ ⟦**AS** *type*⟧ ⟦, *variable*⟦( )⟧ ⟦**AS** *type*⟧⟧...

Remarks The argument *variable* is the module-level variable the procedure will use. It is either an array name followed by ( ), or a variable name. The **AS** *type* clause can be used to indicate the variable's type, which can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, fixed-length string (**STRING** * length), **CURRENCY**, or a user-defined type.

By using either the **SHARED** statement in a **SUB** or **FUNCTION** procedure, or the **SHARED** attribute with **COMMON** or **DIM** in the module-level code, you can use variables in a procedure without passing them as parameters. The **SHARED** attribute used with **COMMON** or **DIM** shares variables among all procedures in a module, while the **SHARED** statement shares variables between a single procedure and the module-level code.

Note The **SHARED** statement shares variables only within a single compiled module. It does not share variables with programs in the Quick library or with procedures compiled separately and linked to the program. The **SHARED** statement shares variables only between the module-level code and a **SUB** or **FUNCTION** procedure in the same module.

The **SHARED** statement can appear only in a **SUB** or **FUNCTION** procedure. For more information, see Chapter 2, "SUB and FUNCTION Procedures" in the *Programmer's Guide*.

See Also COMMON, DIM, SUB

Example

The following example calls a **SUB** procedure named `Convert` that converts the input decimal number to its string representation in the given new base. The string `N$` is shared by the procedure and the main program.

```
DEFINT A-Z
DO
   INPUT "Decimal number (input number <= 0 to quit): ",Decimal
   IF Decimal <= 0 THEN EXIT DO
   INPUT "New base: ",Newbase
   N$ = ""
   PRINT Decimal "base 10 equals ";
   DO WHILE Decimal > 0
      CALL Convert (Decimal,Newbase)
      Decimal = Decimal\Newbase
   LOOP
   PRINT N$ " base" Newbase
   PRINT
LOOP

SUB Convert (D,Nb) STATIC
SHARED N$
   ' Take the remainder to find the value of the current
   ' digit.
   R = D MOD Nb
   ' If the digit is less than ten, return a digit (0-9).
   ' Otherwise, return a letter (A-Z).
   IF R < 10 THEN Digit$ = CHR$(R+48) ELSE Digit$ = CHR$(R+55)
   N$ = RIGHT$(Digit$,1) + N$
END SUB
```

SHELL Function

Action Returns an integer value that is the OS/2 process ID for the shelled process.

Syntax **SHELL**(*commandstring$*)

Remarks The argument *commandstring$* is the name of the OS/2 process to be executed, along with any command-line arguments that the executed process requires. The command string is required.

The **SHELL** function is used only under OS/2 protected mode.

Like the **SHELL** statement, the OS/2 protected-mode **SHELL** function permits a BASIC program to execute another process. In addition, the **SHELL** function allows the BASIC program to continue execution without waiting for the child process to terminate, and returns the process ID of the child process. The process ID is used by some of the OS/2 API functions (which can be called from BASIC).

See Also **SHELL** Statement

Example The following example uses the **SHELL** function to execute three OS/2 commands:

```
ON ERROR GOTO ErrHandler
Ext$ = CHR$(34) + " EXE" + CHR$(34)
' The child process does: DIR | FIND " EXE" | SORT
Child$ = "dir | find " + Ext$ + "| sort"
ProcessID = SHELL(Child$)
PRINT "Process ID is:" ProcessID
END

ErrHandler:
    SELECT CASE ERR
        CASE 73
            PRINT : PRINT "You cannot use the SHELL function in DOS."
        CASE ELSE
    END SELECT
    END
```

SHELL Statement

Action Suspends execution of the BASIC program, runs a .COM, .EXE, .BAT, or .CMD program, or a DOS or OS/2 command, and resumes execution of the BASIC program at the statement following the **SHELL** statement.

Syntax **SHELL** ⟦*commandstring*⟧

Remarks The argument *commandstring* must be a valid string expression that contains the name of a program to run, and any program arguments.

Any .COM file, .EXE file, .BAT program, .CMD program, DOS command, or OS/2 command that runs under the **SHELL** statement is called a "child process." Child processes are executed by the **SHELL** statement, which loads and runs a copy of COMMAND.COM (for DOS) or CMD.EXE (for OS/2) with the /C option.

The /C option allows any parameters in *commandstring* to be passed to the child process. It also allows redirection of standard input and output, and execution of built-in commands such as DIR and PATH.

The program name in *commandstring* can have any extension you wish. If no extension is supplied, COMMAND.COM (for DOS) or CMD.EXE (for OS/2) looks for a .COM file, then an .EXE file, and finally, a .BAT file (for DOS) or a .CMD file (for OS/2). If COMMAND.COM or CMD.EXE is not found, BASIC generates the error message `File not found.` BASIC does not generate an error if COMMAND.COM or CMD.EXE cannot find the file specified in *commandstring*, but the operating system generates an error message.

COMMAND.COM or CMD.EXE treat as program parameters any text separated from the program name by at least one blank. BASIC remains in memory while the child process is running. When the child process finishes, BASIC continues.

If you omit the argument *commandstring*, **SHELL** gives you a new shell (COMMAND.COM for DOS or CMD.EXE for OS/2). You then can enter operating-system commands at the prompt. Use the **EXIT** command to return to BASIC.

Note If you are using OS/2, make sure that the SET COMSPEC configuration command in your CONFIG.SYS file specifies the path for the CMD.EXE file. If the path for CMD.EXE is not set, BASIC generates an error message when **SHELL** searches for the CMD.EXE file.

See Also **SHELL** Function

Examples

The following example shows how a single **SHELL** statement starts up a new COMMAND.COM:

```
' Get a new COMMAND.COM
' Type EXIT at the operating system prompt to return to this program.
SHELL
```

If the name of a any executable file or command is provided as an argument to the **SHELL** statement, that command will be executed as if it had been invoked from the operating-system prompt. When the program is finished, control returns to BASIC. The following use of the **SHELL** statement illustrates how to display a directory listing to check the creation time of a file:

```
SHELL "DIR | MORE"
```

SIGNAL Statements

Action Enable, disable, or suspend event trapping for an OS/2 protected-mode signal.

Syntax

SIGNAL(*n%***) ON**
SIGNAL(*n%***) OFF**
SIGNAL(*n%***) STOP**

Remarks The argument *n%* identifies an OS/2 protected-mode signal. The following table lists the numbers of the protected-mode signals:

Number	Signal
1	Ctrl+C
2	Pipe connection broken
3	Program terminated
4	Ctrl+Break
5	Process flag A
6	Process flag B
7	Process flag C

SIGNAL(*n%***) ON** enables trapping of OS/2 signal *n%*. If an OS/2 protected-mode signal occurs after a **SIGNAL(***n%***) ON** statement, the routine specified in the **ON SIGNAL** statement is executed.

SIGNAL(*n%***) OFF** disables trapping of OS/2 signal *n%*. No trapping takes place until another **SIGNAL(***n%***) ON** statement is executed. Events occurring while trapping is off are ignored.

SIGNAL(*n%***) STOP** suspends trapping of OS/2 signal *n%*. No trapping takes place until a **SIGNAL(***n%***) ON** statement is executed. Events occurring while trapping is suspended are remembered and processed when the next **SIGNAL(***n%***) ON** statement is executed. However, remembered events are lost if **SIGNAL(***n%***) OFF** is executed.

When a signal-event trap occurs (that is, the **GOSUB** is performed), an automatic **SIGNAL STOP** is executed so that recursive traps cannot take place. The **RETURN** operation from the trapping routine automatically performs a **SIGNAL ON** statement unless an explicit **SIGNAL OFF** was performed inside the routine. For more information, see Chapter 9, "Event Handling" in the *Programmer's Guide*.

Note The **SIGNAL** statement is available only for OS/2 protected mode.

See Also ON *event*

Example The following example uses **ON SIGNAL** to trap an event in the OS/2 operating system:

```
PRINT "This program traps Ctrl+Break. Press Q to quit."
EVENT ON

' Set up the signal-event trap and enable.
ON SIGNAL(4) GOSUB CtrlBreak
SIGNAL(4) ON

' Wait until the signal event occurs.
DO : LOOP UNTIL UCASE$(INKEY$) = "Q"
PRINT "'Q' pressed - Program terminating normally."
END

CtrlBreak:
    PRINT "A SIGNAL(4) event occurred."
    PRINT "Press 'Q' to quit."
    RETURN
```

SIN Function

Action Returns the sine of an angle given in radians.

Syntax SIN(*x#*)

Remarks The argument *x#* can be of any numeric type.

The sine of an angle in a right triangle is the ratio between the length of the side opposite the angle and the length of the hypotenuse.

SIN is calculated in single precision if *x#* is an integer or single-precision value. If you use any other numeric data type, **SIN** is calculated in double precision.

To convert values from degrees to radians, multiply the angle (in degrees) times $\pi/180$ (or .0174532925199433). To convert a radian value to degrees, multiply it by $180/\pi$ (or 57.2957795130824). In both cases, $\pi \approx 3.141593$.

See Also ATN, COS, TAN

Example The following example plots the graph of the polar equation $r = 1 + \sin(n * \theta)$. This figure is sometimes known as a cardioid, owing to its resemblance to a heart when n equals 1.

```
CLS
CONST PI = 3.141593
SCREEN 1 : COLOR 1,1       ' Medium resolution, blue background.
WINDOW (-3,-2)-(3,2)       ' Convert screen to Cartesian coordinates.
INPUT "Number of petals = ", N
CLS
PSET (1,0)                 ' Set initial point.
FOR Angle = 0 TO 2 * PI STEP .02
   R = 1 + SIN(N * Angle)   ' Polar equation for "flower."
   X = R * COS(Angle)      ' Convert polar coordinates to
   Y = R * SIN(Angle)      ' Cartesian coordinates.
   LINE -(X,Y)             ' Draw line from previous point to new point.
NEXT
END
```

SLEEP Statement

Action Suspends execution of the calling program.

Syntax **SLEEP** [[*seconds&*]]

Remarks The optional argument *seconds&* determines the number of seconds to suspend the program. The **SLEEP** statement suspends the program until one of the following events occurs:

- The time period in the argument *seconds&* has elapsed.
- A key is pressed.
- An enabled event occurs.

A BASIC event is one you can trap with an **ON** *event* statement such as **ON COM** or **ON TIMER**. A BASIC event cannot interrupt a **SLEEP** suspension unless its trapping is active when the event occurs. This means that trapping must have been initialized with an **ON** *event* statement, turned on with an *event* **ON** statement, and not have been disabled with an *event* **OFF** statement or an *event* **STOP** statement.

If *seconds&* is 0 or is omitted, the program is suspended until a key is pressed or an enabled event occurs.

SLEEP responds only to keystrokes that occur after it executes. **SLEEP** ignores characters in the keyboard buffer that were typed before it executed.

See Also **WAIT**

Example The following example suspends execution for 10 seconds. There is no **ON** *event* statement, so the only way to interrupt the suspension before 10 seconds have passed is to press a key.

```
CLS                                 ' Clear the screen.
PRINT "Taking a 10 second nap..."
SLEEP 10
PRINT "Awake!"
END
```

SOUND Statement

Action Generates sound through the speaker.

Syntax **SOUND** *frequency*, *duration%*

Remarks The **SOUND** statement uses the following arguments:

Argument	Description
frequency	The frequency of the sound in cycles per second or hertz. It must be a numeric expression with a value between 37 and 32,767, inclusive.
duration%	The number of system clock ticks the sound lasts. It must be a numeric expression with an unsigned integer value between 0 and 65,535, inclusive. There are 18.2 clock ticks per second, regardless of CPU speed.

If *duration%* is 0, any current **SOUND** statement that is running in the background is turned off. If no **SOUND** statement is running, a duration of zero has no effect.

Note The **SOUND** statement is not available in OS/2 protected mode.

See Also **PLAY** (Music)

Example The following example produces a rising and descending glissando:

```
FOR I = 440 TO 1000 STEP 5
   SOUND I, I/1000
NEXT
FOR I = 1000 TO 440 STEP -5
  SOUND I, I/1000
NEXT
```

SPACE$ Function

Action Returns a string of spaces of length *n*.

Syntax **SPACE$**(*n*)

Remarks The expression *n* specifies the number of spaces you want in the string. It is rounded to an integer and must be between 0 and 32,767, inclusive.

See Also **LSET, PRINT USING, RSET, SPC, STRING$**

Example The following example demonstrates use of the **SPACE$** function:

```
CLS          ' Clear the screen.
FOR I=1 TO 5
   X$=SPACE$(I)
   PRINT X$;I
NEXT I
```

Output

```
 1
  2
   3
    4
     5
```

SPC Function

Action Skips a specified number of spaces in a **PRINT**, **LPRINT**, OR **PRINT #** statement, starting at the current print position.

Syntax **SPC**(*n%*)

Remarks **SPC** can only be used with **PRINT**, **LPRINT**, or **PRINT #** statements. The argument *n%* is a number between 0 and 32,767, inclusive, that is combined with the width of the output line to determine the number of blank characters to print.

A semicolon (;) is assumed to follow the **SPC** function. For example, the following two print statements are equivalent:

```
PRINT SPC(10) FixLen1$; SPC(10) FixLen2$; SPC(10) FixLen3$
PRINT SPC(10); FixLen1$; SPC(10); FixLen2$; SPC(10); FixLen3$
```

Note that the **SPC** function does more than move the text cursor to a new print position. For screen output it also overwrites any existing characters on a display screen with blanks. The *n%* blank characters are printed starting at the current print position.

The leftmost print position on an output line is always 1; to have any effect, the value of *n%* must be greater than or equal to 1.

The rightmost print position is the current line width of the output device (which can be set with the **WIDTH** statement).

The behavior of an **SPC** function depends on the relationship between three values: *n%*, the output-line print position when the **SPC** function is executed, and the current output-line width:

- If *n%* is greater than the output-line width, **SPC** calculates *n%* **MOD** width and generates the number of blanks indicated by that calculation, starting at the current print position.
- If the difference between the current print position and the output-line width is less than *n%* (or *n%* **MOD** width), the **SPC** function skips to the beginning of the next line and generates a number of blanks equal to *n%* – (width – current print position).

See Also **SPACE$**, **TAB**

Example

The following example demonstrates the use of the **SPC** statement to insert a number of spaces within a printed line using either the **PRINT** statement or the **LPRINT** statement:

```
CLS   ' Clear the screen.
PRINT "The following line is printed using standard screen print"
PRINT "zones."
PRINT : PRINT "Column 1","Column 2","Column 3","Column 4","Column 5"
PRINT : PRINT
PRINT "The next line is printed using the SPC(n%) statement to achieve"
PRINT "the same results."
PRINT
PRINT "Column 1"; SPC(6); "Column 2"; SPC(6); "Column 3";
PRINT SPC(6); "Column 4"; SPC(6); "Column 5"
```

Output

```
The following line is printed using standard screen print
zones.

Column 1      Column 2      Column 3      Column 4      Column 5


The next line is printed using the SPC(n%) statement to achieve
the same results.

Column 1      Column 2      Column 3      Column 4      Column 5
```

SQR Function

Action Returns the square root of a numeric expression.

Syntax **SQR**(*numeric-expression*)

Remarks The argument *numeric-expression* must be greater than or equal to 0.

SQR is calculated in single precision if *numeric-expression* is an integer or single-precision value. If you use any other numeric data type, **SQR** is calculated in double precision.

Example The following example uses the **SQR** function to plot the graph of `y = sqr(abs(x))` for $-9 \le x \le 9$:

```
SCREEN 1 : COLOR 1            ' Low-resolution color graphics mode.
WINDOW (-9,-.25)-(9,3.25)     ' Convert screen to Cartesian coordinates.
LINE (-9,0)-(9,0)             ' Draw x-axis.
LINE (0,-.25)-(0,3.25)        ' Draw y-axis.
FOR x = -9 TO 9
    LINE(x,.04)-(x,-.04)      ' Put tick marks on x-axis.
NEXT x
FOR y = .25 TO 3.25 STEP .25
    LINE (-.08,y)-(.12,y)     ' Put tick marks on y-axis.
NEXT y
PSET (-9,3)                   ' Plot the first point of function.
FOR x = -9 TO 9 STEP .25
    y = SQR(ABS(x))           ' SQR argument cannot be negative.
    LINE -(x,y),2             ' Draw a line to the next point.
NEXT x
```

SSEG Function

Action Returns the segment address of a string (or 0 if the argument is a null string).

Syntax SSEG(*stringvariable$*)

Remarks The argument *stringvariable$* is the string variable for which you want the segment address. It can be a simple string variable or a single element of a string array. You cannot use fixed-length string arguments.

SSEG returns the segment address for strings. It is typically used in mixed-language programming to obtain far addresses before passing far strings to procedures written in other languages.

SSEG usually is used with far strings but also can be used with strings stored in DGROUP (in which case it returns DGROUP's address). The offset of a string can be found by using the **SADD** function.

In OS/2 protected mode, the **SSEG** function returns the selector of the specified string variable.

See Also **BLOAD, BSAVE, DEF SEG, PEEK, POKE, SADD, SSEGADD**

Example See the **SADD** function programming example, which uses the **SSEG** function.

SSEGADD Function

Action Returns the far address of a string variable (both segment and offset).

Syntax **SSEGADD**(*stringvariable$*)

Remarks The argument *stringvariable$* is the string variable for which you want an address. It can be a simple string variable or a single element of a string array. You cannot use fixed-length string arguments.

The **SSEGADD** function combines the segment and offset information into one long integer (4 bytes).

SSEG returns the far address for a string. It is typically used in mixed-language programming to obtain far addresses before passing far strings to procedures written in other languages.

SSEGADD usually is used with far strings but also can be used with strings stored in DGROUP.

The offset of a string can be found with the **SADD** function, while the segment of a string can be found with the **SSEG** function.

See Also **BLOAD, BSAVE, DEF SEG, SADD, SSEG, PEEK, POKE**

Example The following example passes a string to a C routine. It uses **SSEGADD** to obtain the memory address of the string.

```
DEFINT A-Z
DECLARE SUB printmessage CDECL (BYVAL farstring AS LONG)
' Create the message as an ASCIIZ string (last character null),
' as required by the C function printf.
a$ = "This is a short example of a message" + CHR$(0)
' Call the C function with pointers
CALL printmessage(SSEGADD(a$))
```

This C routine prints a BASIC far string on the screen:

```
#include <stdio.h>
/* Define a procedure which inputs a string far pointer */
void printmessage (char far *farpointer)
{
   /* print the string addressed by the far pointer */
   printf( "%s\n", farpointer);
}
```

STACK Function

Action Returns a long integer that is the maximum stack size that can be allocated.

Syntax **STACK**

Remarks The **STACK** function can be used with the **STACK** statement to set the stack to the maximum possible size. For example:

```
STACK STACK
```

See Also **CLEAR**, **STACK** Statement

Example See the **FRE** function programming example, which uses the **STACK** function.

STACK Statement

Action Resets the size of the stack.

Syntax **STACK** [[*longinteger&*]]

Remarks The default stack size is 3K for DOS and 3.5K for OS/2. The minimum stack size (if *longinteger&* is 0) is 0.325K for DOS and 0.825K for OS/2.

The argument *longinteger&* is the number of bytes to reserve for the stack. If *longinteger&* is omitted, the stack is reset to its default size. If you request a stack space that is too large, the **STACK** statement allocates as much stack space as possible.

The **STACK** statement is useful in programs that contain recursion and lots of nesting of **SUB** and **FUNCTION** procedures.

The **STACK** statement is allowed only at the module level.

The **STACK** function can be used with the **STACK** statement to set the stack to the maximum possible size. For example:

```
STACK STACK
```

See Also **CLEAR**, **STACK** Function

Example See the **FRE** function programming example, which uses the **STACK** statement.

$STATIC, $DYNAMIC Metacommands

Action Set aside storage for arrays, while program is either compiling (**$STATIC**) or running (**$DYNAMIC**).

Syntax **REM $STATIC**
' $STATIC

REM $DYNAMIC
' $DYNAMIC

Remarks The **$STATIC** metacommand sets aside storage for arrays during compilation. When the **$STATIC** metacommand is used, the **ERASE** statement reinitializes all array values to zero (numeric arrays) or the null string (string arrays) but does not remove the array.

The **$DYNAMIC** metacommand allocates storage for arrays while the program is running. This means that the **ERASE** statement removes the array and frees the memory it took for other uses. You also can use the **REDIM** statement to change the size of an array allocated using **$DYNAMIC**.

The **$STATIC** and **$DYNAMIC** metacommands affect all arrays except implicitly dimensioned arrays (arrays not declared in a **DIM** statement). Implicitly dimensioned arrays are always allocated as if **$STATIC** had been used. All arrays inside a **SUB** or **FUNCTION** procedure are dynamic unless the **STATIC** keyword is included in the **SUB** or **FUNCTION** statement.

Example See the **ERASE** statement programming example, which demonstrates uses of the **$DYNAMIC** and **$STATIC** metacommands.

STATIC Statement

Action Makes simple variables or arrays local to a **DEF FN** function, a **FUNCTION** procedure, or a **SUB** procedure, and preserves values between calls.

Syntax **STATIC** *variable*⟦( )⟧ ⟦**AS** *type*⟧ ⟦, *variable*⟦( )⟧ ⟦**AS** *type*⟧⟧...

Remarks The **STATIC** statement uses the following arguments:

Argument	Description
variable	The variable the procedure will use. It is either an array name followed by ( ), or a variable name.
AS *type*	Declares the type of the variable. The type can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING** (for variable-length strings), **STRING** * *length* (for fixed-length strings), **CURRENCY**, or a user-defined type.

STATIC is a BASIC declaration that makes simple variables or arrays local to a procedure (or a function defined by **DEF FN**) and preserves the variable value between procedure calls.

The **STATIC** statement can appear only in a **SUB** or **FUNCTION** procedure or **DEF FN** function. Earlier versions of BASIC required the number of dimensions in parentheses after an array name. The number of dimensions in BASIC is now optional.

Variables declared in a **STATIC** statement override variables of the same name shared by **DIM** or **COMMON** statements in the module-level code. Variables in a **STATIC** statement also override global constants of the same name.

Usually, variables used in **DEF FN** functions are global to the module; however, you can use the **STATIC** statement inside a **DEF FN** statement to declare a variable as local to only that function.

The differences between the **STATIC** statement, the **STATIC** attribute, and the **$STATIC** metacommand are:

Statement/Attribute	Differences
STATIC attribute on **SUB** and **FUNCTION** statements	Declares default for variables to be static. Variables having the same name as variables shared by module-level code are still shared.
STATIC statement	Makes specific variables static and overrides any variables shared by the module-level code.
$STATIC metacommand	Affects how memory is allocated for arrays.

See Also **DEF FN**, **FUNCTION**, **SUB**

Example The following example searches for every occurrence of a certain string expression in a file and replaces that string with another string. The program also prints the number of substitutions and the number of lines changed.

```
INPUT "Name of file";F1$
INPUT "String to replace";Old$
INPUT "Replace with";Nw$
Rep = 0 : Num = 0
M = LEN(Old$)
OPEN F1$ FOR INPUT AS #1
CALL Extension
OPEN F2$ FOR OUTPUT AS #2
DO WHILE NOT EOF(1)
   LINE INPUT #1, Temp$
   CALL Search
   PRINT #2, Temp$
LOOP
CLOSE
PRINT "There were ";Rep;" substitutions in ";Num;" lines."
PRINT "Substitutions are in file ";F2$
END

SUB Extension STATIC
SHARED F1$,F2$
   Mark = INSTR(F1$,".")
   IF Mark = 0 THEN
      F2$ = F1$ + ".NEW"
   ELSE
      F2$ = LEFT$(F1$,Mark - 1) + ".NEW"
   END IF
END SUB
```

```
SUB Search STATIC
SHARED Temp$,Old$,Nw$,Rep,Num,M
STATIC R
   Mark = INSTR(Temp$,Old$)
   WHILE Mark
      Part1$ = LEFT$(Temp$,Mark - 1)
      Part2$ = MID$(Temp$,Mark + M)
      Temp$ = Part1$ + Nw$ + Part2$
      R = R + 1
      Mark = INSTR(Temp$,Old$)
   WEND
   IF Rep = R THEN
       EXIT SUB
   ELSE
       Rep = R
       Num = Num + 1
   END IF
END SUB
```

STICK Function

Action Returns joystick coordinates.

Syntax **STICK**(*n%*)

Remarks The argument *n%* is a numeric expression whose value is an unsigned integer from 0 to 3 that indicates which joystick coordinate value to return:

Argument	Value returned
0	Indicates the x coordinate of joystick A.
1	Indicates the y coordinate of joystick A.
2	Indicates the x coordinate of joystick B.
3	Indicates the y coordinate of joystick B.

Joystick coordinates range from 1 to 200 in both directions. You must use `STICK(0)` before you use `STICK(1)`, `STICK(2)`, or `STICK(3)` because `STICK(0)` not only returns the x coordinate of joystick A, but also records the other joystick coordinates. These recorded coordinates are returned by calling `STICK(1)`, `STICK(2)`, or `STICK(3)`.

Note The **STICK** function is not available in OS/2 protected mode.

Example See the **STRIG** statements programming example, which uses the **STICK** function.

STOP Statement

Action Terminates the program.

Syntax STOP ⟦*n%*⟧

Remarks The **STOP** statement halts a program and returns the value *n%* to the operating system. The value *n%* can be used by DOS or OS/2 batch files or by non-BASIC programs. If *n%* is omitted, the **STOP** statement returns a value of 0. Untrapped errors and fatal errors set the value of *n%* to –1.

You can place a **STOP** statement anywhere in a program to terminate execution.

In a stand-alone program, **STOP** closes all files and returns to the operating system; in the QBX environment, **STOP** leaves files open and returns to that environment.

If you use the /D, /E, or /X compiler option on the BC command line, the **STOP** statement prints line numbers as follows:

If your program has:	*STOP prints:*
Line numbers	Number of the line where execution stopped.
No line number for **STOP**	Most recent line number.
No line numbers	0

In the QBX environment, **STOP** always returns an error level of 0, even if you specify a different error level.

In some previous versions of BASIC, **STOP** statements were used for debugging. In the current version of BASIC, you do not have to use **STOP** for debugging.

See Also **END, SYSTEM**

Example See the **ON ERROR** statement programming example, which uses the **STOP** statement.

STR$ Function

Action Returns a string representation of the value of a numeric expression.

Syntax **STR$**(*numeric-expression*)

Remarks If *numeric-expression* is positive, the string returned by the **STR$** function contains a leading blank. The **VAL** function complements **STR$**.

See Also **VAL**

Example The following example uses the **STR$** function to convert a number to its string representation and strips out the leading and trailing blanks that BASIC ordinarily prints with numeric output:

```
CLS                            ' Clear the screen.
PRINT "Enter 0 to end."
DO
    INPUT "Find cosine of: ",Num
    IF Num = 0 THEN EXIT DO
    X$ = STR$(Num)
    NumRemBlanks$ = LTRIM$(RTRIM$(X$))
    PRINT "COS(" NumRemBlanks$ ") = " COS(Num)
LOOP
```

Output

```
Enter 0 to end.
Find cosine of: 3.1
COS(3.1) = -.9991351
Find cosine of: 0
```

STRIG Function

Action Returns the status of a joystick trigger.

Syntax **STRIG**(*n%*)

Remarks The **STRIG** function is used to test the joystick trigger status.

The numeric expression *n%* is an unsigned integer between 0 and 7 that indicates the joystick and trigger to check. The following list describes the values returned by the **STRIG**(*n%*) function for different values of *n%*:

Value	Returns
0	–1 if the lower trigger on joystick A was pressed since the last `STRIG(0)` call, 0 if not.
1	–1 if the lower trigger on joystick A is currently down, 0 if not.
2	–1 if the lower trigger on joystick B was pressed since the last `STRIG(2)` call, 0 if not.
3	–1 if the lower trigger on joystick B is currently down, 0 if not.
4	–1 if the upper trigger on joystick A was pressed since the last `STRIG(4)` call, 0 if not.
5	–1 if the upper trigger on joystick A is currently down, 0 if not.
6	–1 if the upper trigger on joystick B was pressed since the last `STRIG(6)` call, 0 if not.
7	–1 if the upper trigger on joystick B is currently down, 0 if not.

For more information on joystick-event trapping, see the entry for **ON** *event*. You cannot use the **STRIG** function inside a joystick-event trap because the trigger information used by the **STRIG** function will not be available for **ON STRIG**.

In previous versions of BASIC, the statement **STRIG ON** enabled testing of the joystick triggers; **STRIG OFF** disabled joystick-trigger testing. The current version of BASIC ignores these statements. (The old **STRIG ON** and **STRIG OFF** statements are different from the **STRIG**(*n%*) **ON**, **STRIG**(*n%*) **OFF**, and **STRIG**(*n%*) **STOP** statements. The **STRIG** statements enable, disable, and inhibit trapping of joystick events.)

Note The **STRIG** function is not available in OS/2 protected mode.

See Also **ON** *event*, **STRIG** Statements

Example See the **STRIG** statements programming example, which uses the **STRIG** function.

STRIG Statements

Action Enable, disable, or suspend trapping of joystick activity.

Syntax **STRIG(***n%***) ON**
STRIG(*n%***) OFF**
STRIG(*n%***) STOP**

Remarks The argument *n%* is the trigger number as defined in the following table:

n	Trigger	Joystick
0	Lower	First
2	Lower	Second
4	Upper	First
6	Upper	Second

STRIG(*n%***) ON** enables joystick-event trapping. If joystick trigger *n%* is pressed after a **STRIG(***n%***) ON** statement, the routine specified in the **ON STRIG** statement is executed.

The **STRIG(***n%***) OFF** statement disables joystick-event trapping. No joystick event trapping takes place until a **STRIG(***n%***) ON** statement is executed. Events occurring while trapping is off are ignored. However, remembered events are lost if **STRIG(***n%***) OFF** is executed.

The **STRIG(***n%***) STOP** statement suspends joystick-event trapping. No joystick-event trapping takes place until a **STRIG(***n%***) ON** statement is executed. Events occurring while trapping is suspended are remembered and processed when the next **STRIG(***n%***) ON** statement is executed. However, remembered events are lost if **STRIG(***n%***) OFF** is executed.

When a joystick-event trap occurs (that is, the **GOSUB** is performed), an automatic **STRIG STOP** is executed so that recursive traps cannot take place. The **RETURN** operation from the trapping routine automatically performs a **STRIG ON** statement unless an explicit **STRIG OFF** was performed inside the routine.

In previous versions of BASIC, the **STRIG ON** statement enabled testing of the joystick triggers; **STRIG OFF** disabled joystick-trigger testing. The current version of BASIC ignores the **STRIG ON** and **STRIG OFF** statements.

For more information, see Chapter 9, "Event Handling" in the *Programmer's Guide*.

Note The **STRIG** statement is not available for OS/2 protected mode.

See Also **ON** *event*, **STRIG** Function

Example

The following example sets up joystick events for two triggers on each of two joysticks. It also displays the current x and y coordinate position of both joysticks. If any trigger is pressed, control is passed to a routine that indicates the state of the trigger; that is, whether the trigger is down or up. This example exercises the **STRIG** statements, the **STRIG** function, the **ON STRIG** statement, the **STICK** function, and the **EVENT** statements.

```
' Turn on event processing.
EVENT ON

' Enable the upper and lower triggers on both joysticks.
STRIG(0) ON 'Lower trigger, joystick A
STRIG(2) ON 'Lower trigger, joystick B
STRIG(4) ON  'Upper trigger, joystick A
STRIG(6) ON  'Upper trigger, joystick B

' Set up joystick-event processing for each trigger.
ON STRIG(0) GOSUB JoyTriggerHandler
ON STRIG(2) GOSUB JoyTriggerHandler
ON STRIG(4) GOSUB JoyTriggerHandler
ON STRIG(6) GOSUB JoyTriggerHandler

LOCATE 22, 6
PRINT "Press a trigger on either joystick to see a complete report."
LOCATE 23, 20
PRINT "Press ANY keyboard key to end the program."

' Infinite loop waiting for a joystick event or keyboard input.
DO
LOOP WHILE INKEY$ = ""

WrapItUp:
   CLS
   STRIG(0) OFF
   STRIG(2) OFF
   STRIG(4) OFF
   STRIG(6) OFF
   END
```

```
JoyTriggerHandler:
    ' Print a label on screen.
    LOCATE 10, 14: PRINT "Joystick A"
    LOCATE 10, 45: PRINT "Joystick B"
    LOCATE 22, 1: PRINT STRING$(80, 32)
    DO
        IF STRIG(1) THEN   ' Trigger 1, Joystick A.
                TriggerStatus$ = "DOWN"
        ELSE
                TriggerStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 16, 10: PRINT "Trigger 1 is "; TriggerStatus$
        EVENT ON

        IF STRIG(3) THEN   ' Trigger 1, Joystick B.
                TriggerStatus$ = "DOWN"
        ELSE
                TriggerStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 16, 42: PRINT "Trigger 1 is "; TriggerStatus$
        EVENT ON

        IF STRIG(5) THEN   ' Trigger 2, Joystick A.
                TriggerStatus$ = "DOWN"
        ELSE
                TriggerStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 18, 10: PRINT "Trigger 2 is "; TriggerStatus$
        EVENT ON

        IF STRIG(7) THEN    ' Trigger 2, Joystick B.
                TriggerStatus$ = "DOWN"
        ELSE
                TriggerStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 18, 42: PRINT "Trigger 2 is "; TriggerStatus$
        EVENT ON
        GOSUB UpdateXY
    LOOP WHILE INKEY$ = ""
    RETURN WrapItUp
```

```
UpdateXY:
    EVENT OFF
    LOCATE 12, 10: PRINT USING "X Position = ###"; STICK(0)
    LOCATE 14, 10: PRINT USING "Y Position = ###"; STICK(1)
    LOCATE 12, 42: PRINT USING "X Position = ###"; STICK(2)
    LOCATE 14, 42: PRINT USING "Y Position = ###"; STICK(3)
    EVENT ON
    RETURN
```

STRING$ Function

Action Returns a string whose characters all have a given ASCII code or whose characters are all the first character of a string expression.

Syntax 1 **STRING$(***m%*,*n%***)**

Syntax 2 **STRING$(***m%*,*stringexpression$***)**

Remarks The **STRING$** function uses the following arguments:

Argument	Description
m%	A numeric expression indicating the length of the string to return.
n%	The ASCII code of the character to use to build the string. It is a numeric expression that BASIC converts to an integer value between 0 and 255, inclusive.
stringexpression$	The string expression whose first character is used to build the return string.

Examples The first example uses **STRING$** to create part of a report heading:

```
Dash$ = STRING$(10,45)
PRINT Dash$;"MONTHLY REPORT";Dash$
```

Output

```
----------MONTHLY REPORT----------
```

This example uses **STRING$** to generate a bar graph:

```
PRINT TAB(7);"Daily Mean Temperature in Seattle" : PRINT
' Read and graph data for each month.
FOR Month = 1 TO 12 STEP 2
    READ Month$, Temp
    ' Print Temp-35 stars.
    PRINT Month$;" +"; STRING$(Temp-35,"*")
    PRINT "    |"
NEXT Month
' Print horizontal line.
PRINT "    +";
FOR X = 1 TO 7
    PRINT "----+";
NEXT X
PRINT
```

```
' Print temperature labels.
FOR X = 4 TO 39 STEP 5
    PRINT TAB(X); X+31;
NEXT X
PRINT

DATA Jan, 40, Mar, 46, May, 56
DATA Jul, 66, Sep, 61, Nov, 46
```

Output

```
Daily Mean Temperature in Seattle

Jan +*****
    |
Mar +***********
    |
May +*********************
    |
Jul +*******************************
    |
Sep +**************************
    |
Nov +***********
    |
    +----+----+----+----+----+----+----+
    35   40   45   50   55   60   65   70
```

StringAddress Routine

Action Used in mixed-language programming to returns the far address of a variable-length string.

Syntax *far-address* = **StringAddress**(*string-descriptor%*);

Remarks The syntax above is for the C language. For MASM, Pascal, and FORTRAN examples, see Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

StringAddress returns the far address of the variable-length string defined by *string-descriptor%*. The argument *string-descriptor%* is the near address of a string descriptor within a non-BASIC routine.

A non-BASIC routine uses **StringAddress** to find the far address of a BASIC variable-length string. Calls to the **StringAddress** routine always are made from a non-BASIC routine to find the address of a string that was transferred to BASIC from the non-BASIC routine.

For example, assume that you have passed a string from a MASM routine to BASIC's string space using **StringAssign**. The descriptor for the string exists at offset `descriptor.` MASM can find the far address of the string data using the following code:

```
lea   ax, descriptor   ; offset of descriptor
push  ax
extrn stringaddress: proc far
call  stringaddress
```

The far address is returned in DX:AX. DX holds the segment and AX holds the offset.

Note If you are not doing mixed-language programming, there usually is no reason to use **StringAddress**. Instead, use **SSEGADD** to find the far address of a variable-length string.

For more information on mixed-language programming with strings, see Chapter 12, "Mixed-Language Programming" and Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

See Also **SSEGADD**, **StringAssign**, **StringLength**, **StringRelease**

Example

The following example shows how to use **StringAddress** and **StringLength**, routines in the BASIC main library.

The program creates a string in BASIC, then passes it to MASM, which uses **StringAddress** to find the data that needs to be printed. It uses **StringLength** to tell the system the length of the string.

```
DEFINT A-Z
' Create a variable-length string.
Message1$ = "This string was sent to MASM for printing."
' Pass it to MASM to be printed.
CALL PrintMessage1(Message1$)
```

The following MASM procedure must be assembled and the .OBJ file linked to the BASIC code listed above.

```
; *************************PrintMessage1************************
; Prints a string passed by BASIC.

   .model    medium, basic    ; Use same model as BASIC.
   .code

; Define procedure with one-word argument.
PrintMessage1   proc   uses ds, descriptor

; Define external (BASIC library) procedures.
   extrn  StringAddress: proc far
   extrn  StringLength: proc far

   mov    ax, descriptor      ; Find length of string.
   push   ax
   call   StringLength
   push   ax                  ; Save length on stack.

   mov    ax, descriptor      ; Go find out the
   push   ax                  ; address of the string
   CALL   StringAddress       ; data.

   mov    ds, dx              ; Address of string.
   mov    dx, ax
   mov    bx, 01              ; Handle of printer.
   pop    cx                  ; Length of string.
   mov    ah, 40h             ; Have DOS print it.
   int    21h
   ret
PrintMessage1   endp

   end
```

StringAssign Routine

Action Used in mixed-language programming to transfer string data from one language's memory space to that of another.

Syntax **StringAssign**(*sourceaddress&*,*sourcelength%*,*destaddress&*,*destlength%*);

Remarks The syntax above is for the C language. However, the order of the arguments in the syntax follows BASIC, Pascal and FORTRAN calling conventions rather than the C calling convention. (In BASIC, Pascal and FORTRAN, the arguments are passed in the order in which they appear in the source code. In C, the arguments are passed in the reverse order.)

For MASM, Pascal, and FORTRAN examples, see Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

The **StringAssign** routine uses the following arguments:

Argument	Description
sourceaddress&	The far address of the start of string data (if the source is a fixed-length string), or the far address of the string descriptor (if the source is a variable-length string). The *sourceaddress&* is a long integer.
sourcelength%	The string length (if the source is a fixed-length string), or 0 (if the source is a variable-length string). The *sourcelength%* is an integer.
destaddress&	The far address of the start of string data (if the destination is a fixed-length string), or the far address of the string descriptor (if the destination is a variable-length string). The *destaddress&* is a long integer.
destlength%	The string length (if the destination is a fixed-length string), or 0 (if the destination is a variable-length string). The *destlength%* is an integer.

The **StringAssign** routine is used in mixed-language programming to transfer a string from BASIC to a non-BASIC routine or from a non-BASIC routine to BASIC. A typical use is to transfer a string from BASIC to a non-BASIC routine, process the string using the non-BASIC routine, and then transfer the modified string back to BASIC.

Calls to the **StringAssign** routine are usually made from a C, MASM, Pascal, or FORTRAN routine. Seldom, if ever, would you call **StringAssign** from inside a BASIC program.

StringAssign can be used to transfer both near and far strings. Using **StringAssign**, you can write mixed-language programs that are near-string/far-string independent.

MASM, C, Pascal, and FORTRAN deal only with fixed-length strings. When programming in these languages you can use **StringAssign** to create a new BASIC variable-length string and transfer fixed-length string data to it. For example, to transfer a MASM fixed-length string containing the word "Hello" to a BASIC variable-length string, you would use the following data structure:

```
fixedstring  db "Hello"        ; source of data
descriptor   dd 0              ; descriptor for destination
```

The second data element, `descriptor`, is a 4-byte string descriptor initialized to zero. BASIC interprets this to mean that it should create a new variable-length string and associate it with the address labelled `descriptor`.

To create a new BASIC variable-length string and transfer fixed-length data to it, the **StringAssign** routine requires four arguments:

- The far address of the fixed-length string in the MASM data segment.
- The length of the fixed-length string in the MASM data segment.
- The far address of the string descriptor in the MASM data segment.
- 0 (to indicate that the string in the BASIC data segment will be a variable-length string).

The following MASM code pushes these arguments on the stack and calls **StringAssign**:

```
push   ds                 ; segment of fixed-length string
lea    ax, fixedstring    ; offset of fixed-length string
push   ax
mov    ax, 5              ; length of "Hello"
push   ax
push   ds                 ; segment of descriptor
lea    ax, descriptor     ; offset of descriptor
push   ax
mov    ax, 0              ; 0 = variable-length string
push   ax
extrn  stringassign: proc far
call   stringassign
```

When the call to **StringAssign** is made, BASIC will fill in the double-word descriptor with the correct string descriptor.

Note When creating a new variable-length string, you must allocate 4 bytes of static data for a string descriptor as shown above. Allocating the data on the stack will not work.

For more information on mixed-language programming with strings, see Chapter 12, "Mixed-Language Programming" and Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

See Also **StringAddress**, **StringLength**, **StringRelease**

Example The following example shows how to use **StringAssign** and **StringRelease**, routines in the BASIC main library. The program gets a MASM string and prints it on the screen.

```
DEFINT A-Z
' Declare external MASM procedures.
DECLARE FUNCTION MakeString$
DECLARE SUB ReleaseIt

' Get string from MASM and print it.
PRINT MakeString

' Have MASM release the variable-length string it created.
CALL ReleaseIt
```

The following MASM procedure must be assembled and the .OBJ file linked to the BASIC code listed above:

```
; ************************** MakeString ***************************
; Create a fixed-length string and assign it to a BASIC
; variable-length string. Release the string after BASIC uses it.

   .model  medium, basic      ; Use same model as BASIC.
   .data
   ; Create MASM string and a place for BASIC to create a
   ; variable-length string descriptor.
String2      db  "This is a string created by MASM."
Descriptor   dd  0

   .code

   ; Declare BASIC library routines to be used.
   extrn  StringAssign: proc far
   extrn  StringRelease: proc far

MakeString   proc             ; Push arguments:
   push   ds                  ; MASM string segment
   lea    ax, String2         ; MASM string offset
   push   ax
   mov    ax, 33              ; MASM string length
   push   ax
   push   ds                  ; BASIC descriptor segment
   lea    ax, Descriptor
   push   ax                  ; BASIC descriptor offset
   xor    ax, ax              ; Variable-length indicator
   push   ax
   call   StringAssign        ; Assign the string.

   lea    ax, descriptor      ; Return with descriptor
   ret                        ; address in AX.
MakeString   endp

ReleaseIt    proc             ; Routine to release
   lea    ax, Descriptor      ; the string.
   push   ax
   call   StringRelease
   ret
ReleaseIt    endp
   end
```

StringLength Routine

Action Used in mixed-language programming to return the length of a variable-length string.

Syntax *length* = **StringLength**(*string-descriptor%*);

Remarks The syntax above is for the C language. For MASM, Pascal, and FORTRAN examples, see Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

A non-BASIC routine uses **StringLength** to return the number of characters in a BASIC variable-length string.

Calls to the **StringLength** routine are usually made from a non-BASIC routine. The argument *string-descriptor%* is an integer that is the near address of a string descriptor within a non-BASIC routine.

As an example, assume that a MASM routine takes a pointer to a string descriptor as a parameter. MASM can find the length of the string data with the following code:

```
mov   ax, psdparm      ; offset of descriptor
push  ax
extrn stringlength: proc far
call  stringlength
```

The length of the string is returned in AX.

Note If you are not doing mixed-language programming, there is usually no reason to use **StringLength**. Instead, use **LEN** to find the length of a variable-length string.

For more information on mixed-language programming with strings, see Chapter 12, "Mixed-Language Programming" and Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

See Also **LEN**, **StringAddress**, **StringAssign**, **StringRelease**

Example See the **StringAddress** routine programming example, which uses the **StringLength** routine.

StringRelease Routine

Action Deallocates variable-length strings that have been transferred to BASIC's string space from a non-BASIC routine.

Syntax **StringRelease**(*string-descriptor%*);

Remarks The syntax above is for the C language. For MASM, Pascal, and FORTRAN examples, see Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

StringRelease is used in mixed-language programming. **StringRelease** is used by a non-BASIC routine to deallocate variable-length strings that have been transferred to BASIC's string space from the non-BASIC routine.

Calls to the **StringRelease** routine are made from a non-BASIC routine to free up space in BASIC's data area. The argument *string-descriptor%* is an integer that is the near address of a string descriptor within a non-BASIC routine.

BASIC automatically deallocates strings allocated by BASIC. However, strings that have been transferred to BASIC from a non-BASIC routine should be deallocated from the non-BASIC routine using the **StringRelease** routine. (The reason for this is that **StringAssign** transfers strings but not string descriptors. Without the string descriptor, BASIC cannot deallocate the string; the deallocation has to be done from the non-BASIC routine with **StringRelease**.)

As an example, assume that you have passed a string from a MASM routine to BASIC's string space using **StringAssign**. To deallocate this string, assuming a descriptor for the variable-length string exists at offset `descriptor`, the code would be as follows:

```
lea   ax, descriptor   ; offset of descriptor
push  ax
extrn stringrelease: proc far
call  stringrelease
```

Warning Use the **StringRelease** routine only to deallocate variable-length strings that have been transferred to BASIC's string space from a non-BASIC routine. Never use it on strings created by BASIC. Doing so may cause unpredictable results.

For more information on mixed-language programming with strings, see Chapter 12, "Mixed-Language Programming" and Chapter 13, "Mixed-Language Programming with Far Strings" in the *Programmer's Guide*.

See Also **StringAddress, StringAssign, StringLength**

Example See the **StringAssign** routine programming example, which uses the **StringRelease** routine.

SUB Statement

Action Declares the name and the parameters of a **SUB** procedure.

Syntax **SUB** *globalname*⟦(*parameterlist*)⟧ ⟦**STATIC**⟧
⟦*statementblock*⟧
⟦**EXIT SUB**⟧
⟦*statementblock*⟧
END SUB

Remarks The **SUB** statement uses the following arguments:

Argument	Description
globalname	A variable name up to 40 characters long. This name cannot appear in any other **SUB** or **FUNCTION** statement in the same program or the user library. The name cannot include a type-declaration character (**%**, **&**, **!**, **#**, **@**, or **$**).
parameterlist	The list of variables, representing parameters, that will be passed to the **SUB** procedure when it is called. Multiple variables are separated by commas. Parameters are passed by reference, so any change to a parameter's value inside the **SUB** procedure changes its value in the calling program.
STATIC	Indicates that the **SUB** procedure's local variables are to be saved between calls. Without **STATIC**, the local variables are allocated each time the **SUB** procedure is invoked, and the variables' values are lost when the **SUB** returns to the calling program. The **STATIC** attribute does not affect variables that are used in a **SUB** procedure but declared outside the procedure in **DIM** or **COMMON** statements using the **SHARED** statement.
EXIT	Causes an immediate exit from a **SUB**. Program execution continues with the statement after the **CALL** statement.

A **SUB** *parameterlist* argument has the following syntax:

variable[[( )]] [[**AS** *type*]] [[,*variable*[[( )]] [[**AS** *type*]]]]...

Argument	Description
variable	A BASIC variable name. Previous versions of BASIC required the number of dimensions in parentheses after an array name. In the current version of BASIC, the number of dimensions is not required.
AS *type*	The type of the variable: **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **CURRENCY**, **STRING**, or a user-defined type. You cannot use a fixed-length string, or an array of fixed-length strings, as a parameter. However, you can use a simple fixed-length string as an argument in a **CALL** statement; BASIC converts a simple fixed-length string argument to a variable-length string argument before passing the string to a **SUB** procedure.

A **SUB** procedure is a separate procedure, like a **FUNCTION** procedure. However, unlike a **FUNCTION** procedure, a **SUB** procedure cannot be used in an expression.

SUB and **END SUB** mark the beginning and end of a **SUB** procedure. You also can use the optional **EXIT SUB** statement to exit a **SUB** procedure.

SUB procedures are called by a **CALL** statement or by using the procedure name followed by the argument list. See the entry for the **CALL** statement. BASIC **SUB** procedures can be recursive—they can call themselves to perform a given task. See the second example below.

The **STATIC** attribute indicates that all variables local to the **SUB** procedure are static—their values are saved between calls. Using the **STATIC** keyword increases execution speed slightly. **STATIC** usually is not used with recursive **SUB** procedures.

Any **SUB** procedure variables or arrays are considered local to that **SUB** procedure, unless they are explicitly declared as shared variables in a **SHARED** statement. You cannot define **SUB** procedures, **DEF FN** functions, or **FUNCTION** procedures inside a **SUB** procedure.

Note You cannot use **GOSUB**, **GOTO**, or **RETURN** to enter or exit a **SUB** procedure.

See Also

CALL (BASIC), **DECLARE** (BASIC), **SHARED**, **STATIC** Statement

Examples

See the **CALL** (BASIC) statement programming example, which uses the **SUB** statement.

SWAP Statement

Action Exchanges the values of two variables.

Syntax **SWAP** *variable1*, *variable2*

Remarks Any type of variable can be swapped (integer, long, single precision, double precision, string, currency, or record). However, if the two variables are not exactly the same data type, BASIC generates the error message `Type mismatch.`

Example The following example sorts the elements of a string array in descending order using a shell sort. It uses **SWAP** to exchange array elements that are out of order.

```
' Sort the word list using a shell sort.
Num% = 4
Array$(1) = "New York"
Array$(2) = "Boston"
Array$(3) = "Chicago"
Array$(4) = "Seattle"
Span% = Num% \ 2
DO WHILE Span% > 0
   FOR I% = Span% TO Num% - 1
      J% = I% - Span% + 1
      FOR J% = (I% - Span% + 1) TO 1 STEP -Span%
         IF Array$(J%) <= Array$(J% + Span%) THEN EXIT FOR
         ' Swap array elements that are out of order.
         SWAP Array$(J%), Array$(J% + Span%)
      NEXT J%
   NEXT I%
   Span% = Span% \ 2
LOOP
CLS
FOR I% = 1 TO Num%
    PRINT Array$(I%)
NEXT I%
END
```

SYSTEM Statement

Action Closes all open files and returns control to the operating system.

Syntax **SYSTEM** [[*n%*]]

Remarks

The **SYSTEM** statement returns control to the operating system and returns the value *n%* to the operating system. The value *n%* can be used by DOS or OS/2 batch files or by non-BASIC programs. If *n%* is omitted, the **SYSTEM** statement returns a value of 0. Untrapped errors and fatal errors set the value of *n%* to –1.

In a stand-alone program, **SYSTEM** returns to the operating system and closes all files; in the QBX environment, **SYSTEM** stops program execution and closes all files.

Note A program containing a **SYSTEM** statement exits to the operating system if run from the QBX command line with the /RUN option. Entering a **SYSTEM** statement in the Immediate window terminates BASIC.

BASICA **END** and **SYSTEM** are distinct in BASICA, but act identically in the current version of BASIC.

See Also **END**, **STOP**

Example See the **ON ERROR** statement programming example, which uses the **SYSTEM** statement.

TAB Function

Action Moves the text cursor to a specified print position when used in the **PRINT**, **PRINT USING**, **LPRINT**, **LPRINT USING**, and **PRINT #** statements.

Syntax **TAB**(*column%*)

Remarks The argument *column%* is a numeric expression that is the column number of the new print position.

The leftmost print position is always 1. The rightmost position is the current line width of the output device (which can be set with the **WIDTH** statement).

This is an example of using **TAB** in the **PRINT** function:

```
PRINT  TAB(10); City$; TAB(40); State$; TAB(45); Zip$
```

The behavior of **TAB** depends on the relationship between three values: *column%*, the current print position on the current output line when the **TAB** function is executed, and the current output-line width:

- If the current print position on the current line is greater than *column%*, **TAB** skips to *column%* on the next output line.
- If *column%* is less than 1, **TAB** moves the print position to column 1.
- If *column%* is greater than the output-line width, **TAB** calculates:
  print position = *column%* **MOD** width.
- If the calculated print position is less than the current print position, the cursor jumps to the next line at the calculated print position. If the calculated print position is greater than the current print position, the cursor moves to the calculated print position on the same line.

See Also **LPRINT**, **PRINT**, **SPACE$**, **SPC**

Example In the following example **TAB** is also used with the **PRINT** statement to center a text string on an 80-column screen.

```
CLS
INPUT "What is your full name "; Name$
PRINT
' Get the length of Name$, divide by 2, and subtract from screen
' center point.
PRINT TAB(40 - (LEN(Name$) \ 2)); Name$
END
```

TAN Function

Action Returns the tangent of the angle *x*, where *x* is in radians.

Syntax **TAN**(*x*)

Remarks The tangent of an angle in a right triangle is the ratio between the length of the side opposite an angle and the length of the side adjacent to it.

TAN is calculated in single precision if *x* is an integer or single-precision value. If you use any other numeric data type, **TAN** is calculated in double precision.

To convert values from degrees to radians, multiply the angle (in degrees) times $\pi/180$ (or .0174532925199433). To convert a radian value to degrees, multiply it by $180/\pi$ (or 57.2957795130824). In both cases, $\pi \approx 3.141593$.

BASICA In BASICA, if **TAN** overflows, the interpreter generates an error message that reads `Overflow`, returns machine infinity as the result, and continues execution.

If **TAN** overflows, BASIC does not display machine infinity and execution halts (unless the program has an error-handling routine).

See Also **ATN**, **COS**, **SIN**

Example The following example computes the height of an object using the distance from the object and the angle of elevation. The program draws the triangle produced by the base and the computed height.

```
SCREEN 2                          ' CGA screen mode.

INPUT "LENGTH OF BASE: ",Baselen
INPUT "ANGLE OF ELEVATION (DEGREES,MINUTES): ",Deg,Min

Ang = (3.141593/180) * (Deg + Min/60)      ' Convert to radians.
Height = Baselen * TAN(Ang)                ' Calculate height.
PRINT "HEIGHT =" Height
Aspect = 4 * (200 / 640) / 3 ' Screen 2 is 640 x 200 pixels.
H = 180 - Height
B = 15 + (Baselen / Aspect)
LINE (15,180)-(B,180)        ' Draw triangle.
LINE -(B,H)
LINE -(10,180)
LOCATE 24,1 : PRINT "Press any key to continue...";
DO
LOOP WHILE INKEY$=""
```

TEXTCOMP Function

Action Compares two strings as they would be compared by ISAM.

Syntax **TEXTCOMP**(*a$*, *b$*)

Remarks The arguments *a$* and *b$* are two strings. **TEXTCOMP** compares their relative order as they would be sorted by an ISAM index and returns an integer value of 1, 0, or -1:

- -1 is returned when *a$* compares less than *b$*.
- 0 is returned when *a$* compares equal to *b$*.
- 1 is returned when *a$* compares greater than *b$*.

Only the first 255 characters are considered in the comparison.

The comparison is not case sensitive ("A" is the same as "a"), and any trailing spaces are removed. **TEXTCOMP** uses the International Sorting Tables for sorting international characters. For more information, see Appendix E, "International Sorting Tables."

See Also **SetFormatCC**; Appendix E, "International Character Sort Order Tables"

Example The following example uses **TEXTCOMP** to compare titles in a table named BookStock and then prints each title that begins with the word QuickBASIC. Because the comparison performed by **TEXTCOMP** is not case-sensitive, all variations of titles whose first word is QuickBASIC will be printed.

```
'$INCLUDE: 'booklook.bi'              ' RecStruct structure.
DIM BigRec AS RecStruct               ' Define record variable.
OPEN "books.mdb" FOR ISAM Books "BookStock" AS 1      ' Open the ISAM file.
SETINDEX 1, "TitleIndexBS"            ' Set the index.

' Make the first record with "quickbasic" as first word of title current.
SEEKGE 1, "quickbasic"
IF NOT EOF(1) THEN                    ' Quit if no match.
   DO
      IF EOF(1) THEN END              ' Quit if end of table.
      RETRIEVE 1, BigRec.Inventory    ' Fetch record into BigRec.
      ' Compare retrieved record to "quickbasic" in same way comparisons are
      ' done by ISAM. If they don't compare equal, quit program.
      IF TEXTCOMP(LEFT$(BigRec.Inventory.Title, 10), "quickbasic") <> 0 THEN END
      LPRINT BigRec.Inventory.Title ' Print valid titles to line printer.
      MOVENEXT 1                      ' Make next record current.
   LOOP
ELSE
   PRINT "Sorry, SEEK for '; quickbasic; ' failed in BOOKS.MDB"
END IF
```

TIME$ Function

Action Returns the current time from the operating system.

Syntax **TIME$**

Remarks The **TIME$** function returns an eight-character string in the form *hh*:*mm*:*ss*, where *hh* is the hour (00–23), *mm* is minutes (00–59), and *ss* is seconds (00–59). A 24-hour clock is used; therefore, 8:00 P.M. is shown as 20:00:00.

To set the time, use the **TIME$** statement.

See Also **TIME$** Statement, **TIMER** Function

Example The following example converts the 24-hour clock used by **TIME$** to 12-hour output:

```
T$ = TIME$
Hr = VAL(T$)
IF Hr < 12 THEN Ampm$ = " AM" ELSE Ampm$ = " PM"
IF Hr > 12 THEN Hr = Hr - 12
PRINT "The time is" STR$(Hr) RIGHT$(T$,6) Ampm$
```

Output

```
The time is 11:26:31 AM
```

TIME$ Statement

Action Sets the time.

Syntax **TIME$**=*stringexpression$*

Remarks The **TIME$** statement changes the system time, which stays changed until you change it again or reboot your computer.

The argument *stringexpression$* must be in one of the following forms:

Form	Description
hh	Sets the hour; minutes and seconds default to 00.
hh:mm	Sets the hour and minutes; seconds default to 00.
hh:mm:ss	Sets the hour, minutes, and seconds.

A 24-hour clock is used, so 8:00 P.M. would be entered as 20:00:00.

This statement complements the **TIME$** function, which returns the current time.

See Also TIME$ Function

Example The following statement sets the current time to 8:00 A.M.:

```
TIME$="08:00:00"
```

TIMER Function

Action Returns the number of seconds elapsed since midnight.

Syntax **TIMER**

Remarks The **TIMER** function can be used with the **RANDOMIZE** statement to generate a random number. It also can be used to time programs or parts of programs.

See Also **RANDOMIZE**

Example The following example searches for the prime numbers from 3 to 10,000 using a variation of the Sieve of Eratosthenes. The **TIMER** function is used to time the program.

```
DEFINT A-Z
CONST UNMARK = 0, MARKIT = NOT UNMARK
DIM Mark(10000)
CLS                  ' Clear screen.
Start! = TIMER
Num = 0
FOR N = 3 TO 10000 STEP 2
   IF NOT Mark(N) THEN
      'PRINT N,      ' To print the primes, remove the
                     ' remark delimiter in front of the
                     ' PRINT statement.
      Delta = 2 * N
      FOR I = 3 * N TO 10000 STEP Delta
         Mark(I) = MARKIT
      NEXT
      Num = Num + 1
   END IF
NEXT
Finish! = TIMER
PRINT
PRINT "Program took"; Finish! - Start!;
PRINT "seconds to find the"; Num; "primes "
END
```

Output

```
Program took .28125 seconds to find the 1228 primes.
```

TIMER Statements

Action Enable, disable, or suspend timer-event trapping.

Syntax

TIMER ON
TIMER OFF
TIMER STOP

Remarks **TIMER ON** enables timer-event trapping. A timer event occurs when *n* seconds have elapsed (as specified in the **ON TIMER** statement). If a timer event occurs after a **TIMER ON** statement, the routine specified in the **ON TIMER** statement is executed.

TIMER OFF disables timer-event trapping. No timer-event trapping takes place until a **TIMER ON** statement is executed. Events occurring while trapping is off are ignored.

TIMER STOP suspends timer-event trapping. No trapping takes place until a **TIMER ON** statement is executed. Events occurring while trapping is off are remembered and processed when the next **TIMER ON** statement is executed. However, remembered events are lost if **TIMER OFF** is executed.

When a timer-event trap occurs (that is, the **GOSUB** is performed), an automatic **TIMER STOP** is executed so that recursive traps cannot take place. The **RETURN** operation from the trapping routine automatically performs a **TIMER ON** statement unless an explicit **TIMER OFF** was performed inside the routine.

For more information, see Chapter 9, "Event Handling" in the *Programmer's Guide*.

See Also **ON** *event*, **TIMER** Function

Example This example uses the **ON TIMER** statement to trap timer events. It draws a polygon every three seconds with a random shape (three to seven sides), size, and location.

```
SCREEN 1
DEFINT A-Z
DIM X(6), Y(6)
TIMER ON                      ' Enable timer-event trapping.
ON TIMER(3) GOSUB Drawpoly    ' Draw a new polygon every three seconds.
PRINT "Press any key to end program"
INPUT "Press <RETURN> to start",Test$

DO
LOOP WHILE INKEY$ = ""        ' End program if any key pressed.

END

Drawpoly:
   CLS                        ' Erase old polygon.
   N = INT(5 * RND + 2)       ' N is random number from 2 to 6.
   FOR I = 0 TO N
      X(I) = INT(RND * 319)   ' Get coordinates of vertices of
      Y(I) = INT(RND * 199)   ' polygon.
   NEXT
   PSET (X(N), Y(N))
   FOR I = 0 TO N
      LINE -(X(I), Y(I)),2    ' Draw new polygon.
   NEXT
   RETURN
```

TRON/TROFF Statements

Action Trace the execution of program statements.

Syntax **TRON**
TROFF

Remarks In the QBX environment, executing a **TRON** statement has the same effect as selecting Trace On from the Debug menu—each statement is highlighted on the screen as it executes.

The **TROFF** statement turns off the program trace.

The **TRON** and **TROFF** statements display line numbers only when compiled with the Debug option or the /D option on the BC command line.

Note The debugging features of the QBX environment make these statements unnecessary.

Example There is no programming example for the **TRON** statement.

TYPE Statement

Action Defines a data type or ISAM table type that contains one or more elements or table columns.

Syntax

TYPE *usertype*
elementname **AS** *typename*
⟦*elementname* **AS** *typename*⟧
.
.
.
END TYPE

Remarks The **TYPE** statement uses the following arguments:

Argument	Description
usertype	A user-defined data type or ISAM table type. The argument *usertype* follows the same rules as a BASIC variable name. In the case of an ISAM table, *usertype* identifies a user-defined table structure.
elementname	An element or table column of the user-defined type. For a data type, *elementname* follows the same rules as a BASIC variable name. For a table type, *elementname* follows the ISAM naming conventions.
typename	A user-defined data type, nested user-defined type (databases only), an array, or one of the following data types: integer, long, single (non-ISAM types only), double, fixed-length string, or currency.

If *usertype* is a table type, any *elementname* arguments are the names of columns in the table. The names must be exact matches to existing column names and must follow the ISAM naming conventions.

Note Line numbers and line labels aren't allowed in **TYPE...END TYPE** blocks.

ISAM names use the following conventions:

- They have no more than 30 characters.
- They use only alphanumeric characters (A-Z, a-z, 0-9).
- They begin with an alphabetic character.
- They include no BASIC special characters.

Note BASIC now supports user-defined types for ISAM tables, the currency data type for dollars-and-cents math and static arrays in user-defined types. Before you can use an ISAM table, you must declare a record type for the records that make up the table. Instances of this type are used to pass records to and from the table.

The following **TYPE** statement illustrates the use of static arrays. The record `StateData` includes the `CityCode` static array, and the record `Washington` has the same structure as `StateData`:

```
TYPE StateData
   CityCode (1 TO 100) AS INTEGER ' Declare a static array.
   County AS STRING * 30
END TYPE
DIM Washington(1 TO 100) AS StateData
```

When you declare a static array within a user-defined type, its dimensions must be declared with numeric constants rather than variables. For efficiency, make sure that arrays within a user-defined type start on even offsets. You can create very large records when you include static arrays within records. Putting one of these records within a **SUB** procedure can use large amounts of stack space.

Strings in user types must be fixed-length strings. String lengths are indicated by an asterisk and a numeric constant. For example, the following line defines an element named `Keyword` in a user-defined type as a string with length 40:

```
TYPE Keyword AS STRING * 40 END TYPE
```

A user-defined type must be declared in a type declaration before it can be used in the program. Although a user-defined type can be declared only in the module-level code, you can declare a variable to be of a user-defined type anywhere in the module, even in a **SUB** or **FUNCTION** procedure.

Use the **DIM**, **REDIM**, **COMMON**, **STATIC**, or **SHARED** statements to declare a variable to be of a user-defined type.

The keyword **REM** cannot be used as a field name in a **TYPE** statement. The text that follows **REM** is treated as a comment.

Note If you have defined a table to have columns A, B, C, and D, you can use a user-defined type that has only the columns you need (any subset of A, B, C, and D).

See Also **OPEN**

Example See the **PUT** statement (file I/O) programming example, which uses the **TYPE** statement.

UBOUND Function

Action Returns the upper bound (largest available subscript) for the indicated dimension of an array.

Syntax **UBOUND**(*array* [[,*dimension%*]])

Remarks The **UBOUND** function is used with the **LBOUND** function to determine the size of an array. **UBOUND** takes the following arguments:

Argument	Description
array	The name of the array variable to be tested.
dimension%	An integer ranging from 1 to the number of dimensions in *array*; indicates which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second dimension, and so on. This argument is optional for one-dimensional arrays.

UBOUND returns the values listed below for an array with the following dimensions:

```
DIM A(1 TO 100, 1 TO 3, -3 TO 4)
```

Invocation	Value returned
`UBOUND(A,1)`	100
`UBOUND(A,2)`	3
`UBOUND(A,3)`	4

You can use the shortened syntax **UBOUND**(*array*) for one-dimensional arrays because the default value for *dimension%* is 1. Use the **LBOUND** function to find the lower limit of an array dimension.

See Also **DIM, LBOUND, OPTION BASE**

Example

The following example shows how **LBOUND** and **UBOUND** can be used together in a **SUB** procedure to determine the size of an array passed to the procedure by a calling program.

```
DECLARE SUB PRNTMAT (A!())
DIM A(0 TO 3, 0 TO 3)
FOR I% = 0 TO 3
   FOR J% = 0 TO 3
      A(I%, J%) = I% + J%
   NEXT J%
NEXT I%
CALL PRNTMAT(A())
END

SUB PRNTMAT (A()) STATIC
   FOR I% = LBOUND(A, 1) TO UBOUND(A, 1)
   FOR J% = LBOUND(A, 2) TO UBOUND(A, 2)
      PRINT A(I%, J%); " ";
   NEXT J%
   PRINT : PRINT
   NEXT I%
END SUB
```

UCASE$ Function

Action Returns a string expression with all letters in uppercase.

Syntax **UCASE$** (*stringexpression$*)

Remarks The **UCASE$** function takes a string variable, string constant, or string expression as its single argument. **UCASE$** works with both variable- and fixed-length strings.

UCASE$** and **LCASE$ are helpful in making string comparisons that are not case-sensitive.

See Also **LCASE$**

Example This example contains a **FUNCTION** procedure, `YesQues`, that returns a Boolean value depending on how the user responds. The procedure uses **UCASE$** to make a non-case-sensitive test of the user's response.

```
DEFINT A-Z
FUNCTION YesQues(Prompt$,Row,Col) STATIC
   OldRow=CSRLIN
   OldCol=POS(0)
   ' Print prompt at Row, Col.
   LOCATE Row,Col : PRINT Prompt$ "(Y/N):";
   DO
      ' Get the user to press a key.
      DO
         Resp$=INKEY$
      LOOP WHILE Resp$=""
      Resp$=UCASE$(Resp$)
      ' Test to see if it's yes or no.
      IF Resp$="Y" OR Resp$="N" THEN
         EXIT DO
      ELSE
         BEEP
      END IF
   LOOP
   PRINT Resp$;          ' Print the response on the line.
   LOCATE OldRow,OldCol' Move the cursor back to the old position.
   YesQues=(Resp$="Y")  ' Return a Boolean value.
END FUNCTION

DO
LOOP WHILE NOT YesQues("Do you know the frequency?",12,5)
```

UEVENT Statements

Action Enable, disable, or suspend a user-defined event.

Syntax **UEVENT ON**
UEVENT OFF
UEVENT STOP

Remarks The effects of the **UEVENT** statements are like those of other event-trapping statements (such as **COM** or **KEY**). When **UEVENT ON** is executed, the event-trapping routine is enabled. Occurrences of the event trigger execution of the event-handling routine.

When **UEVENT OFF** is executed, the event-trapping routine is disabled. Any occurrences of the event are ignored.

When **UEVENT STOP** is executed, the event-trapping routine is suspended. An event occurrence is remembered, and the event-trapping routine performed as soon as a **UEVENT ON** statement is executed.

When a user-defined event trap occurs (that is, the **GOSUB** is performed), an automatic **UEVENT STOP** is executed so that recursive traps cannot take place. The **RETURN** operation from the trapping subroutine automatically performs a **UEVENT ON** statement unless an explicit **UEVENT OFF** was performed inside the subroutine.

See Also **ON** *event*, **SetUEvent**

Example The following example shows a primitive use of the **ON UEVENT** statement.

```
PRINT "Entering an odd number causes a UEVENT to occur."
ON UEVENT GOSUB Event1
UEVENT ON
DO
    PRINT "Enter a number --> ";
    N = VAL(INPUT$(1)): PRINT N: PRINT
    SELECT CASE N
        CASE 1, 3, 5, 7, 9
            CALL SetUevent    'Odd number was entered, causing UEVENT.
        CASE ELSE
            PRINT "No UEVENT occurred.": PRINT
    END SELECT
    LoopCount = LoopCount + 1
LOOP UNTIL LoopCount = 10

Event1:
    PRINT "Now processing the UEVENT. The odd number was"; N
    RETURN
```

UNLOCK Statement

Action Releases locks applied to parts of a file.

Syntax **UNLOCK** [[#]]*filenumber%* [[, {*record&* | [[*start&*]] **TO** *end&*}]]

Remarks The **UNLOCK** statement is used only after a **LOCK** statement. See the entry for the **LOCK** statement for a complete discussion.

The **UNLOCK** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the file.
record&	The number of the record or byte to be locked. The argument *record&* can be any number between 1 and 2,147,483,647 (equivalent to 2^{31} –1), inclusive. A record can be up to 32,767 bytes in length.
start&	The number of the first record or byte to be locked.
end&	The number of the last record or byte to be locked.

For binary-mode files, *record&*, *start&*, and *end&* represent the number of a byte relative to the beginning of the file. The first byte in a file is byte 1.

For random-access files, *record&*, *start&*, and *end&* represent the number of a record relative to the beginning of the file. The first record is record 1.

Warning Be sure to remove all locks with an **UNLOCK** statement before closing a file or terminating your program. Failing to remove locks produces unpredictable results.

The arguments to **LOCK** and **UNLOCK** must match exactly.

Do not use **LOCK** and **UNLOCK** on devices or ISAM tables.

See Also **LOCK...UNLOCK**

Example See the **LOCK...UNLOCK** statement programming example, which uses the **UNLOCK** statement.

UPDATE Statement

Action Adds an updated record to an ISAM table, overwriting the current record.

Syntax **UPDATE** [[#]]*filenumber%*,*recordvariable*

Remarks The **UPDATE** statement uses the following arguments:

Argument	Description
filenumber%	The number used in the **OPEN** statement to open the table.
recordvariable	A variable of the user-defined type *tabletype* that was specified in the **OPEN** statement. It is the record that will overwrite the current record.

UPDATE replaces the current record with *recordvariable*. It remains the current record.

Use the **RETRIEVE** statement to fetch the current record and place its data into *recordvariable*. You can change the data in *recordvariable*, then use the **UPDATE** statement to update the current record with the changes you've made. Use **INSERT** to add a record to a table without overwriting the current record.

If the values passed to *recordvariable* do not match the record strucure in the user-defined type, BASIC generates the error message `Type Mismatch.` The record structure includes the names and types of columns or fields.

BASIC removes trailing spaces from strings used in an update.

See Also **INSERT**; **RETRIEVE**; **SEEKGT**, **SEEKGE**, **SEEKEQ**

Example See the programming example for the **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements, which uses the **UPDATE** statement.

VAL Function

Action Returns the numeric value of a string of digits.

Syntax **VAL**(*stringexpression$*)

Remarks The argument *stringexpression$* is a sequence of characters that can be interpreted as a numeric value. The **VAL** function stops reading the string at the first character that it cannot recognize as part of a number. The **VAL** function also strips blanks, tabs, and line feeds from the argument string. For example, the code below returns the value 1615198:

```
VAL("    1615 198th Street")
```

See Also **STR$**

Example The following example prints the names and addresses of people with specific telephone area codes:

```
' This part of the program builds a sample data file.
OPEN "PHONE.DAT" FOR OUTPUT AS #1
CLS
RESTORE
READ FuName$, ACPhone$
I = 0
DO WHILE UCASE$(FuName$) <> "END"
   I = I + 1
   WRITE #1, FuName$, ACPhone$
   READ FuName$, ACPhone$
   IF FuName$ = "END" THEN EXIT DO
LOOP
CLOSE #1
DATA "Bob Hartzell ","206-378-3223"
DATA "Alice Provan ","213-884-9700"
DATA "Alex Landow ","213-456-3111"
DATA "Walt Riley ","503-248-0048"
DATA "Georgette Gump ","213-222-2222"
DATA "END",0,0,0,0,0
```

```
' This part of the program demonstrates the VAL function.
INPUT "Search for which area (206, 213, or 503): ", Targetarea
OPEN "PHONE.DAT" FOR INPUT AS #1
DO WHILE NOT EOF(1)
   INPUT #1, Nm$, Phonenum$
   'VAL reads everything up to the first non-numeric
   'character ("-" in this case).
   Area = VAL(Phonenum$)
   IF Area = Targetarea THEN
      PRINT
      PRINT Nm$;
      PRINT Phonenum$
   END IF
LOOP
CLOSE
KILL "PHONE.DAT"
END
```

VARPTR$ Function

Action Returns a string representation of a variable's address for use in **DRAW** and **PLAY** statements.

Syntax **VARPTR$**(*variablename*)

Remarks The argument *variablename* is the name of a variable in the program. If *variablename* is an array element, dimensions for the array must be specified before **VARPTR$** is used. The array must consist of variable-length strings.

Note To guarantee correct results, use the value returned by **VARPTR$** immediately after invoking the function.

BASICA In this version of BASIC, **VARPTR$** must be used in the **DRAW** and **PLAY** statements to execute substrings containing variables. BASICA supports both the **VARPTR$** syntax and the syntax containing just the variable name.

See Also **DRAW**; **PLAY** Statement (Music); **VARPTR**, **VARSEG**

Example See the **PLAY** function programming example, which uses the **VARPTR$** function.

VARPTR, VARSEG Functions

Action Return the address of a variable or string descriptor.

Syntax **VARPTR**(*variablename*)
VARSEG(*variablename*)

Remarks The argument *variablename* can be any BASIC variable, including a record variable or record element.

VARPTR and **VARSEG** return the address of a variable (for numeric variables) or the address of a string descriptor (for string variables) as indicated in the following table:

For a variable of type:	VARSEG returns:	VARPTR returns:
Numeric	Segment address of variable	Offset address of variable
String	Segment address of string descriptor (near strings)	Offset address of string descriptor (near strings)
Name not previously defined	Segment address of new variable (**VARSEG** also creates the new variable)	Offset address of new variable (**VARPTR** also creates the new variable)

When *variablename* is a numeric variable, the **VARPTR** function returns an unsigned integer (the offset of the variable within its segment). When *variablename* is a numeric variable, the **VARSEG** function returns an unsigned integer (the segment address of the variable). When *variablename* is a near-string variable, **VARSEG** and **VARPTR** return a string-descriptor address that contains the length of the string and the address at which it is stored.

If *variablename* is not defined before **VARPTR** or **VARSEG** is called, the variable is created and its address is returned.

VARPTR and **VARSEG** are often used with **Absolute**, **BLOAD**, **BSAVE**, **Interrupt**, **PEEK**, **POKE**, or when passing arrays to procedures written in other languages. When using **VARPTR** or **VARSEG** to get the address of an array, use the first element of the array as the argument:

```
DIM A(150)
.
.
.
ArrAddress=VARPTR(A(1))
```

You can use **VARPTR** alone to get the address of a variable stored in DGROUP. You must use both **VARPTR** and **VARSEG** to get the complete address of a variable stored in far memory.

When programming with OS/2 protected mode, the **VARSEG** function returns the selector of the specified variable or array.

Note Because many BASIC statements move variables in memory, use the values returned by **VARPTR** and **VARSEG** immediately after the functions are used.

It is not meaningful to use **VARSEG** and **VARPTR** for far strings, since the format of the far strings' string descriptor is different from the string descriptor for near strings. See the entry for **StringAddress** for information on locating the address of a far string's string descriptor. See the entries for **SSEG**, **SADD**, and **SSEGADD** for information on locating the segment and offset addresses of far strings.

You can no longer use **VARPTR** to get the address of a file's buffer. Use the **FILEATTR** function to get information about a file.

Programs written in earlier versions of BASIC that used **VARPTR** to access numeric arrays may no longer work. You must now use a combination of **VARPTR** and **VARSEG**. For example, the following QuickBASIC Version 3.0 fragment no longer works correctly:

```
DIM Cube(675)
.
.
.
BSAVE "graph.dat",VARPTR(Cube(1)),2700
```

The fragment could be rewritten as follows to work with the current version of BASIC:

```
DIM Cube(675)
.
.
.
' Change segment to segment containing Cube.
DEF SEG=VARSEG(Cube(1))
BSAVE "graph.dat",VARPTR(Cube(1)),2700
DEF SEG   ' Restore BASIC segment.
```

The **VARSEG** function, combined with **VARPTR**, replaces the PTR86 subprogram used in previous versions of BASIC.

VARPTR and VARSEG and Expanded Memory Arrays

Do not pass expanded memory arrays to non-BASIC procedures. If you start QBX with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays less than 16K in size.
- Fixed-length string arrays less than 16K in size.
- User-defined-type arrays less than 16K in size.

If you want to pass expanded memory arrays to non-BASIC procedures, first start QBX without the /Ea switch. (Without the /Ea switch, no arrays are stored in expanded memory.)

For more information on using expanded memory, see "Memory Management for QBX" in *Getting Started.*

See Also

DEF SEG, PEEK, POKE, SADD, VARPTR$

Example

The following example illustrates how to use the **VARPTR** and **VARSEG** functions in a **CALL** statement to pass a BASIC array to a C routine.

```
DEFINT A-Z
DECLARE SUB AddArr CDECL (BYVAL Offs, BYVAL Segm, BYVAL Num)
DIM A(1 TO 100) AS INTEGER
' Fill the array with the numbers 1 to 15.
FOR I = 1 TO 15
   A(I) = I
NEXT I
'
' Call the C function. AddArr expects a far address (segment
' and offset). Because CDECL puts things on the stack from
' right to left, put the offset ( VARPTR(A(1)) ) first in the
' list, followed by the segment ( VARSEG(A(1)) ).
'
CALL AddArr(VARPTR(A(1)), VARSEG(A(1)), 15)
'
' Print the modified array.
FOR I = 1 TO 15
   PRINT A(I)
NEXT I
END
```

This C routine increments values of a BASIC array. It must be compiled and the .OBJ file linked to the BASIC code above.

```
/* Add one to the first num elements of array arr.*/
void far addarr(arr,num)
int far *arr;
int num;
{
   int i;
   for(i=0;i<num;i++) arr[i]++;
}
```

VIEW Statement

Action Defines screen limits for graphics output.

Syntax **VIEW** [[[[**SCREEN**]] (*x1!*,*y1!*)-(*x2!*,*y2!*) [[,[[*color&*]] [[,*border&*]]]]]]

Remarks The list below describes the parts of the **VIEW** statement:

Part	Description
SCREEN	Specifies that coordinates of subsequent graphics statements are absolute to the screen, not relative to the viewport. Only graphics within the viewport are plotted. When **SCREEN** is omitted, all points are plotted relative to the viewport (*x1!* and *y1!* are added to the coordinates before plotting the point).
(*x1!*,*y1!*)-(*x2!*,*y2!*)	Indicates a rectangular area on the screen. The arguments *x1!*, *y1!*, *x2!*, and *y2!* are numeric expressions that are the coordinates of diagonally opposite corners of the viewport. The argument (*x1!*,*y1!*)-(*x2!*,*y2!*) must be within the physical bounds of the screen in the current screen mode.
color&	A numeric expression that specifies the color with which to fill the viewport. If you omit *color&*, the viewport area is not filled (it has the same color as the screen background color).
border&	Any numeric expression for *border&* causes a line to be drawn around the viewport. The value of border determines the color of the line. If you omit *border&*, no line is drawn around the viewport.

The **VIEW** statement defines a viewport or "clipping region," which is a rectangular section of the screen to which graphics output is limited. If **VIEW** is used with no arguments, the entire screen is defined as the viewport. **RUN** and **SCREEN** also define the entire screen as the viewport.

See Also **PRINT**, **SCREEN** Statement, **WINDOW**

Examples See the **WINDOW** statement programming example, which uses the **VIEW** statement.

VIEW PRINT Statement

Action Sets the boundaries of the screen text viewport.

Syntax **VIEW PRINT** [[*topline%* **TO** *bottomline%*]]

Remarks The argument *topline%* is the number of the upper line in the text viewport; *bottomline%* is the number of the lower line.

Without *topline%* and *bottomline%* parameters, the **VIEW PRINT** statement initializes the whole screen area as the text viewport. The number of lines in the screen depends on the screen mode and whether or not the /H option was used when BASIC was started. For more information, see the entry for the **WIDTH** statement, and Chapter 3, "File and Device I/O" in the *Programmer's Guide*.

Statements and functions that operate within the defined text viewport include **CLS, INPUT**, **LOCATE**, **PRINT**, the **SCREEN** function, and **WRITE**.

See Also **CLS, LOCATE, PRINT, SCREEN** Function, **SCREEN** Statement, **WIDTH**

Example The following example draws random circles in a graphics viewport and prints in a text viewport. The graphics viewport is cleared after 30 circles have been drawn. The program clears the text viewport after printing to it 45 times.

```
RANDOMIZE TIMER
SCREEN 1
' Set up a graphics viewport with a border.
VIEW (5, 5)-(100, 80), 3, 1
' Set up a text viewport.
VIEW PRINT 12 TO 24
' Print a message on the screen outside the text viewport.
LOCATE 25, 1: PRINT "Press any key to stop."
Count = 0
DO
   ' Draw a circle with a random radius.
   CIRCLE (50, 40), INT((35 - 4) * RND + 5), (Count MOD 4)
   ' Clear the graphics viewport every 30 times.
   IF (Count MOD 30) = 0 THEN CLS 1
   PRINT "Hello. ";
   ' Clear the text viewport every 45 times.
   IF (Count MOD 45) = 0 THEN CLS 2
   Count = Count + 1
LOOP UNTIL INKEY$ <> ""
```

WAIT Statement

Action Suspends program execution while monitoring the status of a machine input port.

Syntax **WAIT** *portnumber*, *and-expression%* [[,*xor-expression%*]]

Remarks The **WAIT** statement uses the following arguments:

Argument	Description
portnumber	An unsigned integer expression between 0 and 65,535, inclusive, that is the number of the machine input port.
and-expression%	An integer expression that is repeatedly combined with the bit pattern received at the port, using an **AND** operator; when the result is nonzero, the **WAIT** statement stops monitoring the port, and program execution resumes with the next program statement.
xor-expression%	Can be used to turn bits on and off in the received bit pattern before the **AND** operation is applied.

The **WAIT** statement is an enhanced version of the **INP** function; it suspends execution until a specified bit pattern is input from an input port, using the following four steps:

1. The data byte read from the port is combined, using an **XOR** operation, with *xor-expression%*. If *xor-expression%* is omitted, it is assumed to be 0.
2. The result is combined with *and-expression%* using an **AND** operation.
3. If the result is zero, the first two steps are repeated.
4. If the result is nonzero, execution continues with the next program statement.

It is possible to enter an infinite loop with the **WAIT** statement if the input port fails to develop a non-zero bit pattern. In this case, you must manually restart the machine.

The following example program line illustrates the syntax of the **WAIT** statement:

```
WAIT HandShakePort, 2
```

This statement will cause BASIC to do an **AND** operation on the bit pattern received at the DOS I/O port `HandShakePort` with the bit pattern represented by 2 (00000010).

Note The **WAIT** statement is not available in OS/2 protected mode.

See Also **INP, OUT**

Example The following example demonstrates the use of the **WAIT** statement.

```
' Open and close the port at the proper baud rate.
OPEN "COM1:9600,N,8,1,BIN" FOR RANDOM AS #1
CLOSE #1
PortNum% = &H3F8 'COM1
NonPrintCharMask% = 96
' Wait until there's some activity on COM1.
' Mask out characters less than 32 for the first input character.
LOCATE 12, 15: PRINT "Waiting for a printable input character. "
WAIT PortNum%, NonPrintCharMask%
' Once a printable character comes in, execution continues.
DO
    'Get a character from the port and evaluate.
    A% = INP(PortNum%)
    SELECT CASE A%
        CASE 7
            Character$ = "   BELL    "
        CASE 8
            Character$ = "BACK SPACE "
        CASE 9
            Character$ = "    TAB    "
        CASE 13
            Character$ = "    CR     "
        CASE 32
            Character$ = "   SPACE   "
        CASE 127
            Character$ = "    DEL    "
        CASE IS < 31, IS > 127
            Character$ = "unprintable"
        CASE ELSE
            Character$ = SPACE$(5) + CHR$(A%) + SPACE$(5)
    END SELECT
    LOCATE 12, 15
    ' Report the input character.
    PRINT "Numeric Value                     ASCII Character"
    LOCATE 14, 20
    PRINT A%; TAB(50); Character$
    IF A% > 127 THEN STOP
    LOCATE 23, 25
    PRINT "Press ANY key to quit."
LOOP WHILE INKEY$ = ""
END
```

WHILE...WEND Statement

Action Executes a series of statements in a loop, as long as a given condition is true.

Syntax **WHILE** *condition*

.

.

.

WEND

Remarks The argument *condition* is a numeric expression that BASIC evaluates as true (nonzero) or false (zero).

If *condition* is true (that is, if it does not equal zero), then any intervening statements are executed until the **WEND** statement is encountered. BASIC then returns to the **WHILE** statement and checks *condition*. If it is still true, the process is repeated. If it is not true (or if it equals zero), execution resumes with the statement following the **WEND** statement.

Note BASIC's **DO...LOOP** statement provides a more powerful and flexible loop-control structure.

WHILE...WEND loops can be nested to any level. Each **WEND** matches the most recent **WHILE**. When BASIC encounters an unmatched **WHILE** statement, it generates the error message `WHILE without WEND`. If BASIC encounters an unmatched **WEND** statement, it generates the error message `WEND without WHILE`.

Note Do not branch into the body of a **WHILE...WEND** loop without executing the **WHILE** statement. This may cause run-time errors or program problems that are difficult to locate.

See Also **DO...LOOP**

Example

The following example performs a bubble sort on the array `A$`. Assigning the variable `Exchange` a non-zero value (true) forces one pass through the **WHILE...WEND** loop (this construction is unnecessary with **DO...LOOP**). When there are no more swaps, all elements of `A$` are sorted, `Exchange` is false (equal to zero), and the program continues execution with the line following the **WEND** statement.

```
' Bubble sort of array A$.
CONST FALSE = 0, TRUE = -1
DIM I AS INTEGER
DIM A$(4)
A$(1) = "New York"
A$(2) = "Boston"
A$(3) = "Chicago"
A$(4) = "Seattle"
Max = UBOUND(A$)
Exchange = TRUE          ' Force first pass through the array.
WHILE Exchange           ' Sort until no elements are exchanged.
   Exchange = FALSE
   ' Compare the array elements by pairs. When two are exchanged,
   ' force another pass by setting Exchange to TRUE.
   FOR I = 2 TO Max
      IF A$(I - 1) > A$(I) THEN
         Exchange = TRUE
         SWAP A$(I - 1), A$(I)
      END IF
   NEXT
WEND
CLS
FOR I = 1 TO 4
   PRINT A$(I)
NEXT I
END
```

WIDTH Statements

Action Assign an output-line width to a device (such as a printer) or to a file, or change the number of columns and lines displayed on the screen.

Syntax
WIDTH ⟦*screenwidth%*⟧ ⟦*, screenheight%*⟧
WIDTH {*#filenumber%* | *device$*}*, width%*
WIDTH LPRINT *width%*

Remarks The different forms of the **WIDTH** statements are explained in the following list:

Syntax	Description
WIDTH ⟦*screenwidth%*⟧ ⟦*, screenheight%*⟧	Sets number of columns and lines to display on the screen. The value of *screenwidth%* must be either 40 or 80; default is 80. The table below has details about the value of *screenheight%*.
WIDTH *#filenumber%, width%*	Sets to *width%* the output-line width of an output device already opened as a file (for example, LPT1 or CONS). The argument *filenumber%* is the number used to open the file with the **OPEN** statement. The number sign (#) in front of *filenumber%* is not optional. This form permits altering the width while a file is open, because the statement takes effect immediately.
WIDTH *device$,width%*	Sets to *width%* the line width of *device* (a device filename). The *device* should be a string expression (for example, `"LPT1:"`). Note that he *width%* assignment is deferred until the next **OPEN** statement affecting the device. The assignment does not affect output for an already open file.
WIDTH LPRINT *width%*	Sets to *width%* the line width of the line printer, for use by subsequent **LPRINT** statements. Equivalent to the statement form: `WIDTH "LPT1:",` *width%*

Not all of the argument values are valid in every case; it depends on the installed display adapter and the screen mode established by the most recently executed **SCREEN** statement. The value of *screenheight%* may be 25, 30, 43, 50, or 60, depending on the display adapter used and the screen mode.

Table 1.21 lists the number of lines that can be displayed when a program is started.

Table 1.21 Screen Height as a Function of Display Adapter and Screen Mode

Adapter[1]	Screen mode[2]	Screen width x height	Color display
MDPA	0	80x25	Monochrome
Hercules	0	80x25	Monochrome
	3	80x25	Monochrome
CGA	0	40x25, 80x25	Color
	1	40x25	Color
	2	80x25	Color
EGA	0	40x25, 40x43 80x25, 80x43	Color
	0	40x25, 40x43 80x25, 80x43	Enhanced Color
	0	40x25, 40x43 80x25, 80x43	Monochrome
	1	40x25	Color or Enhanced Color
	2	80x25	Color or Enhanced Color
	7	40x25	Color or Enhanced Color
	8	80x25	Color or Enhanced Color
EGA	9	80x25, 80x43	Enhanced color
EGA or VGA	10	80x25, 80x43	Monochrome
VGA	0	40x25, 40x43, 40x50	Analog
	0	80x25, 80x43, 80x50	Analog
	1	40x25	Analog
	2	80x25	Analog
	7	40x25	Analog
	8	80x25	Analog
	9	80x25, 80x43	Analog
	10	80x25, 80x43	Analog
	11	80x30, 80x60	Analog
	12	80x30, 80x60	Analog
	13	40x25	Analog

Table 1.21 *Continued*

Adapter[1]	Screen mode[2]	Screen width x height	Color display
MCGA	0	40x25, 80x25	Color or analog
	1	40x25	Color or analog
	2	80x25	Color or analog
	11	80x30, 80x60	Analog
	13	40x25	Analog

1 See the **SCREEN** statement for a description of the different adapters.
2 Screen mode 4 has a screen width and screen height of 80x25 in text format. For a list of the computers supported by BASIC screen mode 4, see the **SCREEN** statement.

See Also

SCREEN Statement, **VIEW PRINT**

Example

The following example demonstrates the effect of the **WIDTH** statement on output:

```
CLS
' Open the port at the proper baud rate.
OPEN "COM1:9600,N,8,1,ASC,LF" FOR OUTPUT AS #1
Test$ = "1234567890"' Set up a test string.
WIDTH #1, 3            ' Change width to 3.
PRINT #1, Test$
```

Output

```
123
456
789
0
```

WINDOW Statement

Action Defines the dimensions of the current graphics viewport window.

Syntax **WINDOW** [[[[**SCREEN**]] (*x1!*,*y1!*) – (*x2!*,*y2!*)]]

Remarks The **WINDOW** statement allows the user to create a customized coordinate system to draw lines, graphs, or objects without being constrained by the physical coordinate values and orientation of the graphics viewport. This is done by redefining the graphics viewport coordinates with the "view coordinates" (*x1!*,*y1!*) and (*x2!*,*y2!*). These view coordinates are single-precision numbers.

WINDOW defines a coordinate system that is mapped to the physical coordinates of the viewport. All subsequent graphics statements use the window coordinates and are displayed within the current viewport. (The size of the viewport can be changed with the **VIEW** statement.)

The **RUN** statement, or **WINDOW** with no arguments, disables the window transformation. (The window coordinates are the same as the viewport coordinates.) The **WINDOW SCREEN** variant inverts the normal Cartesian direction of the y coordinate, so y values go from negative to positive from top to bottom.

Figure 1.1 shows the effects of **WINDOW** and **WINDOW SCREEN** on a line drawn in screen mode 2. Notice the change in the coordinates of the screen corners.

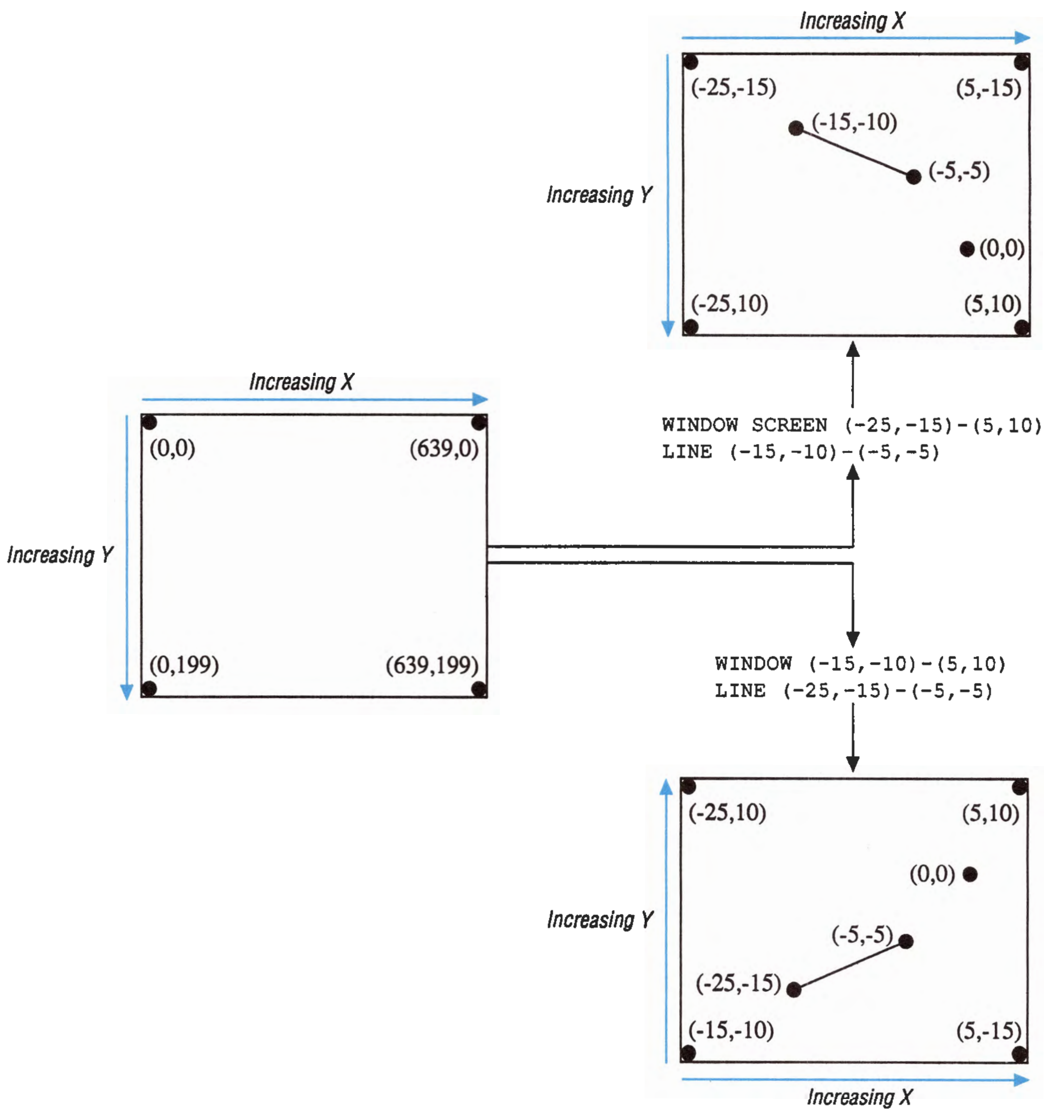

Figure 1.1 WINDOW Contrasted with WINDOW SCREEN

See Also SCREEN Statement, **VIEW**, **WIDTH**

Example The following example uses BASIC graphics statements to generate a fractal. This fractal shows a subset of a class of numbers known as complex numbers; this subset is called the "Mandelbrot Set," named after an IBM researcher. The program uses the **VIEW**, **WINDOW**, **PMAP**, and **PSET** statements.

```
DEFINT A-Z          ' Default variable type is integer.
DECLARE SUB ScreenTest (EM%, CR%, VL%, VR%, VT%, VB%)

' Set maximum number of iterations per point.
CONST MAXLOOP = 30, MAXSIZE = 1000000
CONST FALSE = 0, TRUE = NOT FALSE       ' Boolean constants.
' Set window parameters.
CONST WLeft = -1000, WRight = 250, WTop = 625, WBottom = -625

' Call ScreenTest to find out if this is an EGA machine,
' and get coordinates of viewport corners.
ScreenTest EgaMode, ColorRange, VLeft, VRight, VTop, VBottom

' Define viewport and corresponding window.
VIEW (VLeft, VTop)-(VRight, VBottom), 0, ColorRange
WINDOW (WLeft, WTop)-(WRight, WBottom)

LOCATE 24, 10: PRINT "Press any key to quit.";
XLength = VRight - VLeft
YLength = VBottom - VTop
ColorWidth = MAXLOOP \ ColorRange

' Loop through each pixel in viewport and calculate
'  whether or not it is in the Mandelbrot Set.
FOR Y = 0 TO YLength        ' Loop through every line in the viewport.
   LogicY = PMAP(Y, 3)      ' Get the pixel's logical y coordinate.
   PSET (WLeft, LogicY)     ' Plot leftmost pixel in the line.
   OldColor = 0             ' Start with background color.
   FOR X = 0 TO XLength     ' Loop through every pixel in the line.
      LogicX = PMAP(X, 2)   ' Get the pixel's logical x coordinate.
      MandelX& = LogicX
      MandelY& = LogicY
      ' Do the calculations to see if this point is in
      '  the Mandelbrot Set.
      FOR I = 1 TO MAXLOOP
         RealNum& = MandelX& * MandelX&
         ImagNum& = MandelY& * MandelY&
         IF (RealNum& + ImagNum&) >= MAXSIZE THEN EXIT FOR
         MandelY& = (MandelX& * MandelY&) \ 250 + LogicY
         MandelX& = (RealNum& - ImagNum&) \ 500 + LogicX
      NEXT I
```

```
      ' Assign a color to the point.
      PColor = I \ ColorWidth
      ' If color has changed, draw a line from the
      ' last point referenced to the new point,
      '  using the old color.
      IF PColor <> OldColor THEN
         LINE -(LogicX, LogicY), (ColorRange - OldColor)
         OldColor = PColor
      END IF
      IF INKEY$ <> "" THEN END
   NEXT X
   ' Draw the last line segment to the right edge of
   '  the viewport.
   LINE -(LogicX, LogicY), (ColorRange - OldColor)
NEXT Y
DO : LOOP WHILE INKEY$ = ""
SCREEN 0, 0                ' Restore the screen to text mode,
WIDTH 80                   ' 80 columns.
END

BadScreen:                 ' Error handler that is invoked if
   EgaMode = FALSE         '  there is no EGA graphics card.
   RESUME NEXT

' The ScreenTest SUB procedure tests to see if user has EGA hardware
' EGA hardware with SCREEN 8. If this causes an error, the EM flag
' is set to FALSE, and the screen is set with SCREEN 1. ScreenTest
' also sets values for corners of viewport (VL = left, VR = right,
' VT = top, VB = bottom), scaled with the correct aspect ratio so
' that the viewport is a perfect square.
SUB ScreenTest (EM, CR, VL, VR, VT, VB) STATIC
   EM = TRUE
   ON ERROR GOTO BadScreen
   SCREEN 8, 1
   ON ERROR GOTO 0
   IF EM THEN              ' No error, so SCREEN 8 is OK.
      VL = 110: VR = 529
      VT = 5: VB = 179
      CR = 15              ' 16 colors (0 - 15).
   ELSE                    ' Error, so use SCREEN 1.
      SCREEN 1, 1
      VL = 55: VR = 264
      VT = 5: VB = 179
      CR = 3               ' 4 colors (0 - 3).
   END IF
END SUB
```

WRITE Statement

Action Sends data to the screen.

Syntax **WRITE** [[*expressionlist*]]

Remarks The argument *expressionlist* specifies one or more values to be written on the screen, separated by commas, with double quotation marks around the strings and no spaces around numbers.

If *expressionlist* is omitted, a blank line is written. If *expressionlist* is included, the values of the expressions are written to the screen. The expressions in the list can be numeric and/or string expressions. They must be separated by commas.

When the printed items are written, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage-return-and-line-feed sequence. The **WRITE** statement writes numeric values without leading or trailing spaces.

See Also **PRINT, PRINT USING**

Example The following example shows the difference between the **PRINT** and **WRITE** statements:

```
CLS         ' Clear screen.
A=80 : B=90 : C$="That's all." : D=-1.0E-13
WRITE A,B,C$,D
PRINT A,B,C$,D
```

Output

```
80,90,"That's all.",-1E-13
 80            90            That's all.  -1E-13
```

WRITE # Statement

Action Writes data to a sequential file.

Syntax **WRITE** *#filenumber%*⟦*, expressionlist*⟧

Remarks The *filenumber%* is the number used in the **OPEN** statement to open the file that will contain the data being written. The file must be opened in output or append mode. The expressions in the argument *expressionlist* are string and/or numeric expressions, separated by commas. If you omit *expressionlist*, the **WRITE #** statement writes a blank line to the file.

The **WRITE #** statement, unlike the **PRINT #** statement, inserts commas between items and quotation marks around strings as they are written to the file. You do not have to put explicit delimiters in the list. A new line is inserted once the last item in the list has been written to the file.

If you use **WRITE #** in an attempt to write data to a sequential file restricted by a **LOCK** statement, BASIC generates the error message `Permission denied` unless the error is trapped by the program. All of BASIC's usual error-handling routines can trap and examine this error.

See Also **LOCK**, **OPEN**, **PRINT #**, **WRITE**

Example See the **LINE INPUT #** statement programming example, which uses the **WRITE #** statement.

Part 2

Add-On-Library Reference

Part 2 is an alphabetical reference for each function found in the Date/Time, Financial, and Format add-on libraries. The syntax, options, and action performed by each are described, as well as examples of how to use each in a BASIC program.

Leading off Part 2 is a set of summary tables for the add-on libraries. Each table lists the names of the functions to use for specific tasks, along with the results.

The rest of Part 2 is an alphabetic reference of the Date/Time, Financial, and Format functions.

Add-On-Library Summary Tables

The following groups of functions are summarized here:

- Date/time functions
- Financial functions
- Format functions

The tables on the following pages list the type of task or calculation performed by a function, the function name, and the value that is returned when you use the function. For more information about these functions, see the entries that follow these tables.

These functions are not part of the BASIC language; they are supported by add-on libraries included with Microsoft BASIC. The Setup program can install these libraries for you. For information, see Chapter 1, "Setup" in *Getting Started*. For complete information about creating and using libraries, see Chapter 17, "About Linking and Libraries" in the *Programmer's Guide*.

Date/Time Functions

Table 2.1 lists the functions used in BASIC for converting dates and times to serial numbers and vice versa. These functions are supported by the DTFMT*xx*.LIB libraries or the DTFMTER.QLB Quick library.

Table 2.1 Summary of Date/Time Functions

Task	Function name	Returns
Converting a date to a serial number	**DateSerial#**	Serial number for the specified month, day, and year.
	DateValue#	Serial number for the specified date.
Converting a serial number to a date	**Day&**	Day of the month for a specified serial number.
	Month&	Month for the specified serial number.
	Weekday&	Day of the week for the specified serial number.
	Year&	Year for the specified serial number.
Converting current date and time to serial	**Now#**	Serial number for the current date and time.

Table 2.1 Continued

Task	Function name	Returns
Converting a time to a serial number	**TimeSerial#**	Serial number for the specified hour, minute, and second.
	TimeValue#	Serial number for the specified time.
Converting a serial number to a time	**Hour&**	Hour for the specified serial number.
	Minute&	Minute for the specified serial number.
	Second&	Second for the specified serial number.

Financial Functions

Table 2.2 lists the functions used in BASIC for performing financial calculations. These functions are supported by the FINANC*xx*.LIB libraries or the FINANCER.QLB Quick library.

Table 2.2 Summary of Financial Functions

Calculation	Function name	Returns
Depreciation	**DDB#**	Depreciation of an asset using the double-declining balance method.
	SLN#	Straight-line depreciation for an asset.
	SYD#	Sum-of-years' depreciation for an asset.
Future value	**FV#**	Future value of an investment.
Interest	**IPmt#**	Interest payment for a given period.
	Rate#	Interest rate per period.
Internal rate of return	**IRR#**	Internal rate of return.
	MIRR#	Modified internal rate of return.
Number of payments	**NPer#**	Number of payments for an investment.
Present value	**NPV#**	Net present value of an investment.
	PV#	Present value of an investment.
Principal and interest payments	**Pmt#**	Periodic payment for an investment (including principal and interest).
Principal payments	**PPmt#**	Payment on the principal for an investment for a given period.

Format Functions

Table 2.3 lists the functions used in BASIC for formatting numeric values. These functions are supported by the DTFMT*xx*.LIB libraries or the DTFMTER.QLB Quick library.

Table 2.3 Summary of Format Functions

Task	Function name	Action
Formatting numeric values as strings	**FormatD$**	Takes a double-precision data type and returns a formatted string.
	FormatC$	Takes a currency data type and returns a formatted string.
	FormatI$	Takes an integer data type and returns a formatted string.
	FormatL$	Takes a long-integer data type and returns a formatted string.
	FormatS$	Takes a single-precision data type and returns a formatted string.
	SetFormatCC	Sets the country code used by the **Format*X*$** functions.

DateSerial# Function

Action Returns a serial number that represents the date of the arguments.

Syntax **DateSerial#** (*year%*, *month%*, *day%*)

Remarks The **DateSerial#** function uses the following arguments:

Argument	Description
year%	A year between 1753 and 2078, inclusive.
month%	A month between 1 and 12, inclusive.
day%	A day between 1 and 31, inclusive.

For the argument *year%*, values between 0 and 178, inclusive, are interpreted as the years 1900-2078. For all other *year* arguments, use the complete four-digit year (for example, 1800).

For each of the three arguments, a value out of the specified range generates the error message `Illegal function call.`

You can use negative numbers as arguments. For example, to find the serial number for the date one week before 9/1/89, you could use:

```
DateSerial#(89,9,1-7)
```

For more information, see the topic "Serial Numbers" in this section.

To use **DateSerial#** in the QBX environment, use the DTFMTER.QLB Quick library. To use **DateSerial#** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **DateValue#**, **Day&**, **Month&**, **Now#**, Serial Numbers, **Weekday&**, **Year&**

Example The following example prompts you to enter the last two digits of your birth year.

Birth years in the 20th century are assumed. The year information is used by the **DateSerial#** function to calculate a unique date value for January 1 in the year of your birth. You are then told what day of the week January 1 fell on in that year. Information is displayed using **Year&** and **Weekday&** functions.

To run this example, you must link it with the appropriate DTFMT*xx*.LIB file or use the DTFMTER.QLB Quick library. The following include file also must be present:

```
' $INCLUDE: 'DATIM.BI'

OPTION BASE 1
DEFINT A-Z

DIM DayOfWeek(7) AS STRING

' Initialize the array.
FOR I = 1 TO 7
   READ DayOfWeek$(I)
NEXT I

' Get user input.
INPUT "What are the last two digits of your birth year"; Birthyear

' Calculate January 1st of birth year.
Jan1Date# = DateSerial#(Birthyear, 1, 1)

' Display results.
PRINT "January 1,"; Year&(Jan1Date#); "fell on a ";
PRINT DayOfWeek$(Weekday&(Jan1Date#))

DATA "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday"
DATA "Friday", "Saturday", "Sunday"
```

DateValue# Function

Action Returns a serial number that represents the date of the argument.

Syntax **DateValue#** (*datetext$*)

Remarks The argument *datetext$* is a date between January 1, 1753, and December 31, 2078, inclusive. It can have date formats such as "December 30, 1988", "30/12/88 2:24 AM", and "28-Dec-1988".

If the year part of *datetext$* is omitted, **DateValue#** uses the current year from your computer's system date.

Any time information used in *datetext$* is ignored. If *datetext$* includes time information, **DateValue#** does not return this time information. However, if *datetext$* includes invalid time information, (such as "89:97"), BASIC generates the error message `Illegal function call.`

For more information, see the topic "Serial Numbers" in this section.

To use **DateValue#** in the QBX environment, use the DTFMTER.QLB Quick library. To use **DateValue#** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **DateSerial#**, **Day&**, **Month&**, **Now#**, Serial Numbers, **Weekday&**, **Year&**

Example The following example prompts you to enter your birthday as a string in the form *mm/dd/yyyy*. The string is converted to a unique date value using the **DateValue#** function so that further calculations can be accomplished. A calculation is then made that tells you the date of two weeks before and two weeks after your birth. Date information is displayed using **Weekday&**, **Day&**, **Month&**, and **Year&** functions.

To run this example, you must link it with the appropriate DTFMT*xx*.LIB file or use the DTFMTER.QLB Quick library. The following include file also must be present:

```
' $INCLUDE: 'DATIM.BI'
OPTION BASE 1
DEFINT A-Z
DIM DayOfWeek(7) AS STRING
DIM MonthOfYear(12) AS STRING
' Initialize the arrays.
FOR I = 1 TO 7
   READ DayOfWeek(I)
NEXT I
FOR I = 1 TO 12
   READ MonthOfYear(I)
NEXT I

INPUT "Enter your birth date as MM/DD/YYYY "; Birthdate$

' Convert input date to numbers and create a complete date string.
BDateVal# = DateValue#(Birthdate$) 'Get the unique date value.
BDay$ = DayOfWeek$(Weekday&( BDateVal#)) + " " + MonthOfYear$(Month&(BDateVal#))
BDay$ = BDay$ + STR$(Day&( BDateVal#)) + "," + STR$(Year&(BDateVal#))

' Calculate a date 14 days earlier.
EDateVal# = BDateVal# - 14
' Put an earlier date string together.
EDay$ = DayOfWeek$(Weekday&( EDateVal#)) + " " + MonthOfYear$(Month&(EDateVal#))
EDay$ = EDay$ + STR$(Day&( EDateVal#)) + "," + STR$(Year&(EDateVal#))

' Calculate a date 14 days later.
LDateVal# = BDateVal# + 14
' Put a later date string together.
LDay$ = DayOfWeek$(Weekday&( LDateVal#)) + " " + MonthOfYear$(Month&(LDateVal#))
LDay$ = LDay$ + STR$(Day&( LDateVal#)) + "," + STR$(Year&(LDateVal#))

PRINT "You were born on "; BDay$; "."
PRINT
PRINT "What you may not have been able to immediately calculate was:"
PRINT
PRINT "Two weeks before your birth was "; EDay$; "."
PRINT "Two weeks after your birth was "; LDay$; "."
END

DATA "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday"
DATA "Friday", "Saturday"
DATA "January", "February", "March", "April", "May", "June", "July"
DATA "August", "September", "October", "November", "December"
```

Day& Function

Action Takes a date/time serial number and returns the day of the month.

Syntax **Day&** (*serial#*)

Remarks The **Day&** function returns an integer between 1 and 31, inclusive, that represents the day of the month corresponding to the argument.

The argument *serial#* is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Day&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Day&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Hour&**, **Minute&**, **Month&**, **Now#**, **Second&**, Serial Numbers, **Weekday&**, **Year&**

Example See the **DateValue#** function programming example, which uses the **Day&** function.

DDB# Function

Action Returns depreciation of an asset for a specific period using the double-declining-balance method.

Syntax **DDB#** (*cost#*, *salvage#*, *life#*, *period#*, *status%*)

Remarks The double-declining-balance method computes depreciation at an accelerated rate. Depreciation is highest in the first period and decreases in successive periods.

The **DDB#** function uses the following arguments:

Argument	Description
cost#	The initial cost of the asset.
salvage#	The value at the end of the useful life of the asset.
life#	The useful life of the asset.
period#	The period for which asset depreciation is desired.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

Note All numeric arguments must be positive values.

The arguments *life#* and *period#* must use the same units. If *life#* is given in months, *period#* also must be given in months.

DDB# uses the formula:

Depreciation for a period = ((*cost* – total depreciation from prior periods) * 2) / *life*

To use **DDB#** in the QBX environment, use the FINANCER.QLB Quick library. To use **DDB#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **SLN#**, **SYD#**

Example The following example uses **DDB#** to return the depreciation of an asset for a specific period using the double-declining balance method based on the asset's initial cost, salvage value, and the useful life of the asset.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files also must be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'

CONST YEARMONTHS = 12
CONST DOLLARFORMAT$ = "$###,###,###.00"

DEFDBL A-Z
DIM status  AS INTEGER

' Get user input.
CLS
PRINT "Assume you have a piece of expensive manufacturing equipment."
INPUT "Enter the original cost of the equipment"; Cost
PRINT
PRINT "Salvage value is the value of the asset at the end of its useful life."
INPUT "What is the salvage value of the equipment"; SalvageVal
PRINT
INPUT "What is the useful life of the equipment in months"; Life
IF Life < YEARMONTHS THEN
    PRINT
    PRINT "If the life is less than a year, it is not an asset."
    INPUT "Re-enter the useful life of the equipment in months"; Life
END IF

YearsInLife = Life / YEARMONTHS

' Round up to a whole year.
IF YearsInLife <> INT(Life / YEARMONTHS) THEN
    YearsInLife = INT(YearsInLife + 1)
END IF
```

```
PRINT
PRINT "For what year of the asset's life do you want to calculate ";
INPUT "depreciation "; DeprYear
DO WHILE Depr < YearsInLife AND DeprYear > YearsInLife
    PRINT
    PRINT "The year you enter must be at least 1, and not more than";
    PRINT YearsInLife
    INPUT "Re-enter the year"; DeprYear
LOOP

' Calculate and format the results.
TotalDepreciation = Cost - SalvageVal
PeriodDepreciation = DDB#(Cost, SalvageVal, YearsInLife, DeprYear, status)
MonthDepreciation = PeriodDepreciation / YEARMONTHS

Cost$ = FormatD$(Cost, DOLLARFORMAT$)
SalvageVal$ = FormatD$(SalvageVal, DOLLARFORMAT$)
TotalDepreciation$ = FormatD$(TotalDepreciation, DOLLARFORMAT$)
PeriodDepreciation$ = FormatD$(PeriodDepreciation, DOLLARFORMAT$)
MonthDepreciation$ = FormatD$(MonthDepreciation, DOLLARFORMAT$)

' Examine status to determine success or failure of DDB#.
IF status THEN
    ' If unsuccessful, announce a problem.
    PRINT
    PRINT "There was an error in calculating depreciation."
ELSE
    ' If successful, display the results.
    PRINT
    PRINT "If the original cost of your equipment is "; Cost$
    PRINT "and the salvage value of that equipment is "; SalvageVal$
    PRINT "at the end of its"; Life; "month lifetime, the total "
    PRINT "depreciation is "; TotalDepreciation$; ". "
    PRINT
    PRINT "The depreciation in year"; DeprYear; "is "; PeriodDepreciation$
    PRINT "or "; MonthDepreciation$; " per month."
END IF
```

FormatX$ Functions

Action Formats a numeric value.

Syntax **FormatI$** (*expression#*, *fmt$*)
**FormatL$** (*expression#*, *fmt$*)
**FormatS$** (*expression#*, *fmt$*)
**FormatD$** (*expression#*, *fmt$*)
**FormatC$** (*expression#*, *fmt$*)

Remarks The **Format***X***$** functions return a string that contains the formatted expression. The functions use the following arguments:

Argument	Description
expression#	A numeric expression to be formatted.
fmt$	A string expression comprising BASIC display-format characters that detail how the expression is to be displayed.

There are five separate add-on-library functions, depending on the data type of *expression#*. Use the appropriate function as shown below:

Function	Data type of *expression*
FormatI$	Integer
FormatL$	Long integer
FormatS$	Single precision
FormatD$	Double precision
FormatC$	Currency

If *fmt$* is a null string, BASIC uses the general format (which gives output identical to output with no formatting).

To use the **Format***X***$** functions in the QBX environment, use the DTFMTER.QLB Quick library. To use the **Format***X***$** functions outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The FORMAT.BI header file contains the necessary function declarations for using the **Format*X*$** functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

Available Formats

The **Format*X*$** functions provide a wide range of formats for numeric and date/time data. The following symbols are used to create the formats:

Symbol	Meaning
Null string	Display the number in general format.
0	Digit placeholder. If the number has fewer digits on either side of the decimal point than there are zeros on either side of the decimal point in the format, the extra zeros are displayed. If the number has more digits to the right of the decimal point than there are zeros to the right in the format, the number is rounded to as many decimal places as there are zeros to the right. If the number has more digits to the left of the decimal point than there are zeros to the left in the format, the extra digits are displayed.
#	Digit placeholder. Follows the same rules as for the 0 digit placeholder, except that extra zeros are not displayed if the number has fewer digits on either side of the decimal point than there are number signs (#) on either side of the decimal point.
.	Decimal point. This symbol determines how many digits BASIC displays to the right and left of the decimal point. Depending on the country code set using the **SetFormatCC** routine, BASIC may use the comma as the decimal point. If the format contains only number signs (#) to the left of this symbol, numbers smaller than 1 are begun with a decimal point. To avoid this, you should use 0 as the first digit placeholder to the left of a decimal point instead of #.
%	Percentage. The expression is multiplied by 100 and the percent character (%) is inserted.

,	Thousands separator. BASIC separates thousands by commas (or by periods, depending on the country code set using the **SetFormatCC** routine) if the format contains a comma surrounded by digit placeholders (0 or #). Two adjacent commas, or a comma immediately to the left of the decimal point location (whether there is a decimal specified or not) means "Omit the three digits that fall between these commas, or between the comma and the decimal point, rounding as needed."
E– E+ e– e+	Scientific format. If a format contains one digit placeholder (0 or #) to the right of an E–, E+, e–, or e+, BASIC displays the number in scientific format and inserts an E or e. The number of digit placeholders to the right determines the number of digits in the exponent. Use E– or e– to place a minus sign next to negative exponents. Use E+ or e+ to place a minus sign next to negative exponents and a plus sign next to positive exponents.
: – + $ ( ) Space	Display that literal character. To display characters other than one of these, precede each character with a backslash(\) or enclose the character(s) in double quotation marks ("").
\	Display the next character in the format string. Many characters in the format string have a special meaning and cannot be displayed as literal characters unless they are preceded by a backslash. The backslash is not displayed. This is the same as enclosing the next character in double quotation marks. Examples of such characters are the date- and time-formatting characters (y, m, d, h, s, a, and p) and the numeric-formatting characters (#, 0, %, E, e, comma, and period).
"*abc*"	Display whatever text is inside the double quotation marks. To include a text string in *fmt$*, you must use `CHR$(34)` to enclose the text (34 is the ASCII code for double quotation mark).

Some sample numeric formats are shown below. (These examples all assume the country code is set to 1, United States.)

	Result for:		
Format (*fmt$*)	**5**	**–5**	**.5**
Null string	5	–5	0.5
0	5	–5	1
0.00	5.00	–5.00	0.50
#,##0	5	–5	1
#,##0.00	5.00	–5.00	0.50
$#,##0;($#,##0)	$5	($5)	$1
$#,##0.00;($#,##0.00)	$5.00	($5.00)	$0.50
0%	500%	–500%	50%
0.00%	500.00%	–500.00%	50.00%
0.00E+00	5.00E+00	–5.00E+00	5.00E–01
0.00E–00	5.00E00	–5.00E00	5.00E–01

A number format can have three sections separated by semicolons. The first section formats positive values; the second section formats negative values; and the third section formats zeros.

The following example has two sections: the first section defines the format for positive numbers and zeros; the second section defines the format for negative numbers.

```
$#,##0;($#,##0)
```

You can use from one to three sections:

If you use:	*The result is:*
One section only	The format applies to all numbers.
Two sections	The first section applies to positive numbers and zeros; the second to negative numbers.
Three sections	The first section applies to positive numbers, the second to negative numbers, and the third to zeros.

If you have semicolons with nothing between them, the missing section is printed using the format of the positive value. For example, the following format will display positive and negative numbers using the format in the first section, and "Zero" if the value is zero:

```
"$#,##0;;Z\ero"
```

Date/Time Formats

Date/time serial numbers can be formatted with date/time or numeric formats (since date/time serial numbers are stored as floating-point values).

Date/time formats have the following meanings:

Symbol	Meaning
d dd ddd dddd	Display the day as a number without leading zeros (1–12), as a number with leading zeros (01–12), as an abbreviation (Sun–Sat), or as a full name (Sunday–Saturday).
m mm mmm mmmm	Display the month as a number without leading zeros (1–12), as a number with leading zeros (01–12), as an abbreviation (Jan–Dec), or as a full month name (January–December). If you use m or mm immediately after the h or hh symbol, the minute rather than the month is displayed.
yy yyyy	Display the year as a two–digit number (00–99), or as a four–digit number (1900–2040).
h hh	Display the hour as a number without leading zeros (0–23), or as a number with leading zeros (00–23). If the format contains an AM or PM, the hour is based on the 12-hour clock. Otherwise, the hour is based on the 24-hour clock.
m mm	Display the minute as a number without leading zeros (0–59), or as a number with leading zeros (00–59). The m or mm must appear after an h or hh, or the month is displayed rather than the minute.
s ss	Display the second as a number without leading zeros (0–59), or as a number with leading zeros (00–59).
AM/PM am/pm A/P a/p	Display the hour using the 12-hour clock. AM, am, A, or a is displayed with any time before noon. PM, pm, P, or p is displayed with any time between noon and 11:59 P.M.

The following are examples of date and time formats:

Format	Display
m/d/yy	`2/7/58`
d-mmmm-yy	`7-December-58`
d-mmmm	`7-December`
mmmm-yy	`December-58`
h:mm AM/PM	`8:50 PM`
h:mm:ss AM/PM	`8:50:35 PM`
h:mm	`20:50`
h:mm:ss	`20:50:35`
m/d/yy h:mm	`12/7/58 20:50`

See Also **PRINT**, **PRINT USING**, **STR$**

Example See the **Now#** function programming example, which uses the **FormatD$** function.

FV# Function

Action Returns the future value of an annuity based on periodic, constant payments and a constant interest rate.

Syntax **FV#** (*rate#*, *nper#*, *pmt#*, *pv#*, *type%*, *status%*)

Remarks An annuity is a series of constant cash payments made over a continuous period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **FV#** function uses the following arguments:

Argument	Description
rate#	The interest rate per period. For example, if you get a car loan at an annual rate of 10 percent and make monthly payments, the rate per period would be .10/12, or .0083.
nper#	The number of payment periods in the annuity. For example, if you get a four-year car loan and make monthly payments, your loan has a total number of 4 times 12, or 48, payment periods.
pmt#	The payment to be made each period. It usually contains principal and interest and does not change over the life of the annuity.
pv#	The present value or a lump sum that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type* if payments are due at the end of the period; use 1 if due at the beginning of the period.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *rate#* and *nper#* must use consistent units. For example:

rate#	***nper#***	**Loan description**
.10/12	4*12	Monthly payment, four-year loan, 10 percent annual interest
.10	4	Annual payment, four-year loan, 10 percent annual interest

Note For all arguments, cash you pay out, such as deposits to savings, is represented by negative numbers; cash you receive, such as dividend checks, is represented by positive numbers.

FV# uses the formula:

$$pv * (1+rate)^{nper} + pmt(1+rate*type)*\left(\frac{(1+rate)^{nper}-1}{rate}\right)+fv=0$$

A number of other functions are related to **FV#**:

- **IPmt#** returns the interest payment for an annuity for a given period.
- **NPer#** returns the number of periods (payments) for an annuity.
- **Pmt#** returns the periodic total payment for an annuity.
- **PPmt#** returns the principal payment for an annuity for a given period.
- **PV#** returns the present value of an annuity.
- **Rate#** returns the interest rate per period of an annuity.

To use **FV#** in the QBX environment, use the FINANCER.QLB Quick library. To use **FV#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **IPmt#, NPer#, Pmt#, PPmt#, PV#, Rate#**

Example The following example uses **FV#** to return the future value of an investment. You are prompted to input any required information.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files also must be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST YEARMONTHS = 12
CONST ENDPERIOD = 0
CONST BEGINPERIOD = 1
CONST DOLLARFORMAT$ = "$#,###,##0.00"
CONST PERCENTFORMAT$ = "##.00"
DEFDBL A-Z
DIM APR AS SINGLE
DIM PaymentType AS INTEGER, Status AS INTEGER

' Get user input.
CLS
INPUT "Enter the expected annual percentage rate "; APR
PRINT
INPUT "How much do you plan to invest each month"; Payment
PRINT
DO WHILE PaymentTypeString$ <> "B" AND PaymentTypeString$ <> "E"
    PRINT "Make payments at the beginning or end of the month? ";
    PaymentTypeString$ = UCASE$(INPUT$(1))
LOOP
' Set up the correct payment type.
IF PaymentTypeString$ = "B" THEN
    PaymentType = BEGINPERIOD
    PRINT "Beginning"
ELSE
    PaymentType = ENDPERIOD
    PRINT "End"
END IF
PRINT
INPUT "How long, in months, do you expect to invest"; Period
PRINT
INPUT "What is the current value of this investment"; CurVal
PRINT
```

```
' Calculate and format the results.

'Put APR in proper form
IF APR <= 1 THEN
    PercentageRate$ = FormatS$(APR * 100, PERCENTFORMAT$) + "%"
ELSE
    PercentageRate$ = FormatS$(APR, PERCENTFORMAT$) + "%"
    APR = APR / 100
END IF

Payment$ = FormatD$(Payment, DOLLARFORMAT$)
CurValue$ = FormatD$(CurVal, DOLLARFORMAT$)
FVal = FV#(APR / YEARMONTHS, Period, -Payment, -CurVal, PaymentType, Status)
FutureValue$ = FormatD$(FVal, DOLLARFORMAT$)

' Examine Status to determine success of failure of FV#.
IF Status THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the future value."
ELSE
    ' If successful, display the results.
    PRINT
    PRINT "At "; PercentageRate$; " interest, your "; Payment$; " per month "

    IF PaymentType = BEGINPERIOD THEN
        PRINT "(paid at the beginning of each month), "
    ELSE
        PRINT "(paid at the end of each month), "
    END IF

    IF CurVal > 0 THEN
        PRINT "plus the "; CurValue$; " your investment is currently worth,"
    END IF

    PRINT "will be worth "; FutureValue$; " after"; Period; "months."
END IF
```

Hour& Function

Action Takes a date/time serial number and returns the hour.

Syntax **Hour&** (*serial#*)

Remarks The **Hour&** function returns an integer between 0 (12:00 A.M.) and 23 (11:00 P.M.) that represents the hour of the day corresponding to the argument.

The argument *serial#* is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Hour&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Hour&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&**, **Minute&**, **Month&**, **Second&**, Serial Numbers, **Weekday&**, **Year&**

Example See the **TimeSerial#** function programming example, which uses the **Hour&** function.

IPmt# Function

Action Returns the interest payment for a given period of an annuity based on periodic, constant payments and a constant interest rate.

Syntax **IPmt#** (*rate#*, *per#*, *nper#*, *pv#*, *fv#*, *type%*, *status%*)

Remarks An annuity is a series of constant cash payments made over a continuous period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **IPmt#** function uses the following arguments:

Argument	Description
rate#	The interest rate per period. For example, if you get a car loan at an annual rate of 10 percent and make monthly payments, the rate per period would be .10 divided by 12, or .0083.
per#	The payment period; must be in the range from 1 to *nper#*.
nper#	The number of payment periods in the annuity. For example, if you get a four-year car loan and make monthly payments, your loan has a total number of 4 times 12, or 48, payment periods.
pv#	The present value or a lump sum that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv#	The future value or the cash balance you want sometime in the future after the last payment is made. The future value of a loan, for instance, is 0. As another example, if you think you will need $50,000 in 18 years to pay for your child's education, then $50,000 is the future value.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type%* if payments are due at the end of the period; use 1 if due at the beginning of the period.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *rate#* and *nper#* must use consistent units. For example:

rate#	*nper#*	**Loan description**
.10/12	4*12	Monthly payment, four-year loan, 10 percent annual interest
.10	4	Annual payment, four-year loan, 10 percent annual interest

Note For all arguments, negative numbers equal cash you pay out, and positive numbers equal cash you receive.

A number of other functions are related to **IPmt#**:

- **FV#** returns the the future value of an annuity.
- **NPer#** returns the number of periods (payments) for an annuity.
- **Pmt#** returns the periodic total payment for an annuity.
- **PPmt#** returns the principal payment for an annuity for a given period.
- **PV#** returns the present value of an annuity.
- **Rate#** returns the interest rate per period of an annuity.

To use **IPmt#** in the QBX environment, use the FINANCER.QLB Quick library. To use **IPmt#** outside of the QBX environment, link your program with the appropriate FINANCxx.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	**Compiler options**
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **FV#, NPer#, Pmt#, PPmt#, PV#, Rate#**

Example

This example uses the **IPmt#** function to calculate how much you will pay over the lifetime of a loan where all the payments are of equal value.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files must also be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST YEARMONTHS = 12
CONST ENDPERIOD = 0
CONST BEGINPERIOD = 1
CONST DOLLARFORMAT$ = "$###,###,###.00"
FutureVal = 0
DEFDBL A-Z
DIM Status  AS INTEGER

' Get user input.
CLS
INPUT "Enter the annual percentage rate of your loan"; APR
PRINT
INPUT "What is the principal amount of your loan"; PresentVal
PRINT
INPUT "How many monthly payments do you have to make"; NumPeriods
PRINT
DO WHILE PaymentTypeString$ <> "B" AND PaymentTypeString$ <> "E"
    PRINT "Payments due at the beginning or end of the month? ";
    PaymentTypeString$ = UCASE$(INPUT$(1))
LOOP

' Set up the correct payment type.
IF PaymentTypeString$ = "B" THEN
    PaymentType = BEGINPERIOD
    PRINT "Beginning"
ELSE
    PaymentType = ENDPERIOD
    PRINT "End"
END IF
PRINT

' Put APR in proper form.
IF APR > 1 THEN APR = APR / 100
```

```
FOR Period = 1 TO NumPeriods
    ' Calculate the Interest paid each month.
    InterestPmt = ABS(IPmt#(APR / YEARMONTHS, Period, NumPeriods,_
                  -PresentVal, FutureVal, PaymentType, Status))

    ' Check for an error.
    IF Status THEN
        PRINT "There was a calculation error."
        END
    END IF
    ' Accumulate a total.
    TotalInterest = TotalInterest + InterestPmt
NEXT Period

' Format and display results.
TotalInterest$ = FormatD$(TotalInterest, DOLLARFORMAT$)
PresentVal$ = FormatD$(PresentVal, DOLLARFORMAT$)
TotalPaid$ = FormatD$(TotalInterest + PresentVal, DOLLARFORMAT$)
PRINT
PRINT "You'll pay a total of "; TotalPaid$; " on your "; PresentVal$;
PRINT " loan, of which "; TotalInterest$; " is interest."
```

IRR# Function

Action Returns the internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax **IRR#** (*valuearray#*( ), *valuecount%*, *guess#*, *status%*)

Remarks The internal rate of return is the interest rate received for an investment that consists of payments and receipts that occur at regular intervals.

The **IRR#** function uses the following arguments:

Argument	Description
valuearray#( )	An array of cash-flow values. The argument must contain at least one negative value (a payment) and one positive value (a receipt).
valuecount%	The total number of items in the cash-flow array.
guess#	A value you guess is close to the result of **IRR#**. In most cases you can assume *guess#* to be 0.1 (10 percent). However, if **IRR#** returns a status of 1 (failure), or if the result is not close to what you expected, try different values of *guess#*.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

IRR# uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence. The cash flow for each period does not have to be even as it would be for an annuity.

Note **IRR#** is calculated by iteration. Starting with the value of *guess#*, **IRR#** cycles through the calculation until the result is accurate within .00001 percent. If after 20 tries it can't find a result that works, **IRR#** returns a status of 1 (failure).

Several other functions are related to **IRR#**:

- **MIRR#** gives the internal rate of return where positive and negative cash flows are financed at different rates.
- **NPV#** gives the net present value of an investment based on cash flows that do not have to be constant.
- **Rate#** gives the interest rate per period of an investment.

To use **IRR#** in the QBX environment, use the FINANCER.QLB Quick library. To use **IRR#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **MIRR#, NPV#, Rate#**

Example The following example uses **IRR#** to determine the internal rate of return on cash flow for a series of periods as represented in an array of values. It then uses **NPV#** to determine the net present value for the same cash-flow periods used to determine the internal rate of return.

To run this example, you must link it with the appropriate FINANC*xx*.LIB file or use the FINANCER.QLB Quick library. The following include file must also be present:

```
' $INCLUDE: 'FINANC.BI'
OPTION BASE 1
CONST DOLLARFORMAT$ = "$#,###.##"
CONST PERCENTFORMAT$ = "##.##"
DEFDBL A-Z
DIM Status  AS INTEGER, NumFlows  AS INTEGER
NumFlows% = 6
DIM Values(NumFlows%) AS DOUBLE
Guess = .1
CLS

' Read cash flows into Values# array.
FOR I = 1 TO NumFlows%
    READ Values#(I)
NEXT I

' Calculate the internal rate of return.
ReturnRate = IRR#(Values#(), NumFlows%, Guess, Status%)
```

```
' Examine Status% to determine success or failure of IRR#.
IF Status% THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the internal rate of return."
ELSE
    ' Display the internal rate of return.
    PRINT "You're starting a printing business. You estimate that it will"
    PRINT "cost you $70,000.00 in start-up costs. Your net annual income "
    PRINT "for the next five years is expected to be $12,000, $15,000, "
    PRINT "$18,000, $21,000, and $26,000, respectively."
    PRINT
    PRINT "The internal rate of return for these cash-flow figures is";
    PRINT USING PERCENTFORMAT$; ReturnRate * 100;
    PRINT "%."
    PRINT
    ' Calculate the net present value.
    NetPresentValue = NPV#(ReturnRate, Values#(), NumFlows%, Status%)
    ' Check Status% again.
    IF Status% THEN
        PRINT "There was an error in calculating the net present value."
    ELSE
        PRINT "The net present value of the cash flows used to calculate"
        PRINT "the internal rate of return is effectively ";
        PRINT USING DOLLARFORMAT$; NetPresentValue;
        PRINT "."
    END IF
END IF

' Estimated cost of starting up a printing business (negative cash flow).
DATA -70000

' Net income each year for five successive years (positive cash flow).
DATA 12000,15000,18000,21000,26000
```

Minute& Function

Action Takes a date/time serial number and returns the minute.

Syntax **Minute&** (*serial#*)

Remarks The **Minute&** function returns an integer between 0 and 59, inclusive, that represents the minute corresponding to the argument.

The argument *serial#* is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Minute&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Minute&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&**, **Hour&**, **Month&**, **Now#**, **Second&**, Serial Numbers, **Weekday&**, **Year&**

Example See the **TimeSerial#** function programming example, which uses the **Minute&** function.

MIRR# Function

Action Returns the modified internal rate of return for a series of periodic cash flows (payments and receipts).

Syntax **MIRR#** (*valuearray#*( ), *valuecount%*, *fin-rate#*, *reinv-rate#*, *status%*)

Remarks The modified internal rate of return is the internal rate of return where payments and receipts are financed at different rates. **MIRR#** considers both the cost of the investment (*fin-rate#*) and the interest received on reinvestment of cash (*reinv-rate#*).

The **MIRR#** function uses the following arguments:

Argument	Description
valuearray#( )	An array of cash-flow values. The argument must contain at least one negative value (a payment) and one positive value (a receipt).
valuecount%	The total number of items in the cash-flow array.
fin-rate#	The interest rate paid as the cost of financing.
reinv-rate#	The interest rate received on gains from cash reinvestment.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *fin-rate#* and *reinv-rate#* are percentages expressed as decimal values. For example, 12 percent is expressed as .12.

MIRR# uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence.

MIRR# uses the formula:

$$\left[\frac{-\mathrm{NPV}(\mathrm{rrate},\mathrm{values[positive]}) * (1+rrate)^{n}}{\mathrm{NVP}(\mathrm{frate},\mathrm{values[negative]}) * (1+frate)}\right]^{\frac{1}{n-1}} - 1$$

Two other functions are related to **MIRR#**:

- **IRR#** returns the internal rate of return for an investment.
- **Rate#** returns the interest rate per period of an annuity.

To use **MIRR#** in the QBX environment, use the FINANCER.QLB Quick library. To use **MIRR#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **IRR#**, **Rate#**

Example The following example uses **MIRR#** to determine the modified internal rate of return cash flow for a series of periods as represented in an array of values.

To run this example, you must link it with the appropriate FINANC*xx*.LIB file or use the FINANCER.QLB Quick library. The following include file must also be present:

```
' $INCLUDE: 'FINANC.BI'
OPTION BASE 1
CONST PERCENTFORMAT$ = "##.##"
DEFDBL A-Z
DIM Status  AS INTEGER, NumFlows  AS INTEGER
NumFlows% = 6
DIM Values(NumFlows%) AS DOUBLE
LoanAPR = .1
InvestAPR = .12
' Read cash flows into Values# array.
FOR I = 1 TO NumFlows%
    READ Values#(I)
NEXT I
CLS

' Calculate the internal rate of return.
ReturnRate = MIRR#(Values#(), NumFlows%, LoanAPR, InvestAPR, Status%)
```

```
' Examine Status% to determine success or failure of IRR.
IF Status% THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the internal rate of return."
ELSE
    ' Display the internal rate of return.
    PRINT "Suppose you're a commercial fisherman and you've just completed"
    PRINT "your fifth year of operation. When you started your business,"
    PRINT "you borrowed $120,000 at 10% annual interest to buy a boat."
    PRINT
    PRINT "Your catch yielded $39,000, $30,000, $21,000, $37,000, and $46,000"
    PRINT "for each of the first five years.  During the first 5 years you"
    PRINT "reinvested your profits, earning 12% annually."
    PRINT
    PRINT "The modified internal rate of return for these cash-flow figures"
    PRINT "is ";
    PRINT USING PERCENTFORMAT$; ReturnRate * 100;
    PRINT "%."
    PRINT
END IF

' Cost of buying a commercial fishing boat (negative cash flow).
DATA -120000

' Net income each year for five successive years (positive cash flow).
DATA 39000,30000,21000,37000,46000
```

Month& Function

Action Takes a date/time serial number and returns the month.

Syntax **Month&** (*serial#*)

Remarks The **Month&** function returns an integer between 1 and 12, inclusive, that represents the month corresponding to the argument.

The argument *serial#* is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Month&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Month&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&**, **Hour&**, **Minute&**, **Now#**, **Second&**, Serial Numbers, **Weekday&**, **Year&**

Example See the **DateValue#** function programming example, which uses the **Month&** function.

Now# Function

Action Returns a serial number that represents the current date and time according to your computer's system date and time.

Syntax **Now#**

Remarks For more information, see the topic "Serial Numbers" in this section.

To use **Now#** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Now#** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&**, **Hour&**, **Minute&**, **Month&**, **Second&**, Serial Numbers, **Weekday&**, **Year&**

Example The following example examines the current date and time information contained by the operating system and displays a formatted version of the date and time as returned by the **Now#** function.

To run this example, you must link it with the appropriate DTFMT*xx*.LIB file or use the DTFMTER.QLB Quick library. The following include files must also be present.

```
' $INCLUDE: 'DATIM.BI'
' $INCLUDE: 'FORMAT.BI'

CLS
PRINT "Today's date is "; FormatD$(Now#, "dd-mmm-yy"); "."
PRINT "The current time is "; FormatD$(Now#, "hh:mm:ss AM/PM"); "."
```

Output

```
Today's date is 02-May-1989.
The current time is 2:22:14 PM.
```

NPer# Function

Action Returns the number of periods for an annuity based on periodic, constant payments and a constant interest rate.

Syntax **NPer#** (*rate#*, *pmt#*, *pv#*, *fv#*, *type%*, *status%*)

Remarks An annuity is a series of constant cash payments made over a continuous period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **NPer#** function uses the following arguments:

Argument	Description
rate#	The interest rate per period. For example, if you get a car loan at an annual rate of 10 percent and make monthly payments, the rate per period would be .10 divided by 12, or .0083.
pmt#	The payment to be made each period. It usually contains principal and interest and does not change over the life of the annuity.
pv#	The present value or a lump sum that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv#	The future value or the cash balance you want to be attained sometime in the future after the last payment is made. The future value of a loan, for example is zero. As another example, if you think you will need $50,000 in 18 years to pay for your child's education, the future value is $50,000.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type%* if payments are due ar the end of the period; use 1 if at the beginning of the period.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

Note For all arguments, cash you pay out, such as deposits to savings, is represented by negative numbers; cash you receive, such as dividend checks, is represented by positive numbers.

NPer# uses the formula:

$$pv*(1+rate)^{nper}+pmt(1+rate*type)*\left(\frac{(1+rate)^{nper}-1}{rate}\right)+fv=0$$

A number of other functions are related to **NPer#**:

- **FV#** returns the the future value of an annuity.
- **IPmt#** returns the interest payment for an annuity for a given period.
- **Pmt#** returns the periodic total payment for an annuity.
- **PPmt#** returns the principal payment for an annuity for a given period.
- **PV#** returns the present value of an annuity.
- **Rate#** returns the interest rate per period of an annuity.

To use **NPer#** in the QBX environment, use the FINANCER.QLB Quick library. To use **NPer#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **FV#, IPmt#, Pmt#, PPmt#, PV#, Rate#**

Example The following example uses **NPer#** to determine how many periods, or payments, must be made to pay off a loan.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files also must be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST YEARMONTHS = 12
CONST ENDPERIOD = 0
CONST BEGINPERIOD = 1
DEFDBL A-Z
DIM APR AS SINGLE
DIM Status AS INTEGER

PresentValue = 8000
FutureValue = 0
Payment = 250
APR = .102

' Explain the premise.
CLS
PRINT "You're trying to buy a new car that costs $8000.00. Your budget"
PRINT "only allows you to spend up to $250.00 per month to buy the car"
PRINT "and you don't have a down payment."
PRINT
PRINT "Car loans are normally financed for 12, 24, 36, or 48 months. You'd"
PRINT "like to figure out how long you must finance to stay within your"
PRINT "budget."
PRINT

NumPeriods = NPer#(APR / YEARMONTHS, -Payment, PresentValue, FutureValue,_
                   BEGINPERIOD, Status)

' Examine Status% to determine success of failure of NPer#.
IF Status% THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the number of periods in your loan."
ELSE
    ' If successful, display the results.
    PRINT "Calculations reveal that at exactly $250.00 per month"
    PRINT "you would have to finance for at least"; INT(NumPeriods); "months."
    PRINT
    PRINT "It appears that you'll have to finance for 48 months unless you"
    PRINT "can come up with a $210.00 downpayment."
END IF
```

NPV# Function

Action Returns the net present value of an investment based on a series of periodic cash flows (payments and receipts) and a discount rate.

Syntax **NPV#** (*rate#*, *valuearray#*( ), *valuecount%*, *status%*)

Remarks The net present value of an investment is today's value of a future series of payments and receipts.

The **NPV#** function uses the following arguments:

Argument	Description
rate#	The discount rate over the length of the period, expressed as a decimal.
valuearray#( )	An array of cash-flow values. The argument must contain at least one negative value (a payment) and one positive value (a receipt).
valuecount%	The total number of items in the cash-flow array.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

NPV# uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and income values in the correct sequence. The cash-flow values must be equally spaced in time and occur at the end of each period.

The **NPV#** investment begins one period before the date of the first cash-flow value and ends with the last cash flow in the list.

The **NPV#** calculation is based on future cash flows. If your first cash flow occurs at the beginning of the first period, the first value must be added to the **NPV#** result, not included in the cash-flow values of *valuearray#*( )

The differences between **NPV#** and **PV#** are:

PV# cash flow:	NPV# cash flow:
Begins at the end or the beginning of the period.	Occurs at the end of the period.
Is constant throughout the investment.	Is variable.

NPV# uses the formula:

$$\text{NPV} = \sum_{i=1}^{n} \frac{values_i}{(1+rate)^i}$$

Several other functions are related to **NPV#**:

- **FV#** returns the the future value of an annuity.
- **IRR#** returns the internal rate of return for an investment.
- **PV#** returns the present value of an annuity.

To use **NPV#** in the QBX environment, use the FINANCER.QLB Quick library. To use **NPV#** outside of the QBX environment, link your program with the appropriate FINANCxx.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **FV#, IRR#, PV#**

Example See the **IRR#** function programming example, which uses the **NPV#** function.

Pmt# Function

Action Returns the payment for an annuity based on periodic, constant payments and a constant interest rate.

Syntax **Pmt#** (*rate#*, *nper#*, *pv#*, *fv#*, *type%*, *status%*)

Remarks An annuity is a series of constant cash payments made over a continuous period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **Pmt#** function uses the following arguments:

Argument	Description
rate#	The interest rate per period. For example, if you get a car loan at an annual rate of 10 percent and make monthly payments, the rate per period would be .10 divided by 12, or .0083.
nper#	The number of payment periods in the annuity. For example, if you get a four-year car loan and make monthly payments, your loan has a total number of 4 times 12, or 48, payment periods.
pv#	The present value or a lump sum that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv#	The future value or the cash balance you want to be attained sometime in the future after the last payment is made. The future value of a loan, for example is zero. As another example, if you think you will need $50,000 in 18 years to pay for your child's education, the future value is $50,000.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type%* if payments are due at the end of the period; use 1 if due at the beginning of the period.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *rate#* and *nper#* must use consistent units. For example:

rate#	*nper#*	Loan description
.10/12	4*12	Monthly payment, four-year loan, 10 percent annual interest
.10	4	Annual payment, four-year loan, 10 percent annual interest

Note

For all arguments, cash you pay out, such as deposits to savings, is represented by negative numbers; cash you receive, such as dividend checks, is represented by positive numbers.

Pmt# uses the formula:

$$pv * (1+rate)^{nper} + pmt(1+rate*type)*\left(\frac{(1+rate)^{nper}-1}{rate}\right)+fv=0$$

A number of other functions are related to **Pmt#**:

- **FV#** returns the the future value of an annuity.
- **IPmt#** returns the interest payment for an annuity for a given period.
- **NPer#** returns the number of periods (payments) for an annuity.
- **PPmt#** returns the principal payment for an annuity for a given period.
- **PV#** returns the present value of an annuity.
- **Rate#** returns the interest rate per period of an annuity.

To use **Pmt#** in the QBX environment, use the FINANCER.QLB Quick library. To use **Pmt#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	*Compiler options*
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **FV#, IPmt#, NPer#, PPmt#, PV#, Rate#**

Example The following example uses **Pmt#** to determine the payment for a loan over a prescribed period, at a fixed annual percentage rate. It also uses **PPmt#** to determine what part of the payment is principal. The program subtracts the principal payment from the monthly payment to determine the interest part of the payment.

To run this example, you must link it with the appropriate FINANC*xx*.LIB file or use the FINANCER.QLB Quick library. The following include file also must be present:

```
' $INCLUDE: 'FINANC.BI'
CONST YEARMONTHS = 12
CONST ENDPERIOD = 0
CONST BEGINPERIOD = 1
CONST DOLLARFORMAT$ = "$#,###.##     "
CONST PERCENTFORMAT$ = "##.##"
FutureVal = 0
DEFDBL A-Z
DIM Status  AS INTEGER

' Get user input.
CLS
INPUT "Enter the annual percentage rate of your loan"; APR
PRINT
INPUT "What is the principal amount of your loan"; PresentVal
PRINT
INPUT "How many monthly payments is your loan set up for"; NumPeriods
PRINT
DO WHILE PaymentTypeString$ <> "B" AND PaymentTypeString$ <> "E"
    PRINT "Are your payments due at the beginning or end of the month? ";
    PaymentTypeString$ = UCASE$(INPUT$(1))
LOOP

' Set up the correct payment type.
IF PaymentTypeString$ = "B" THEN
    PaymentType = BEGINPERIOD
    PRINT "Beginning"
ELSE
    PaymentType = ENDPERIOD
    PRINT "End"
END IF
PRINT
```

```
' Calculate and format the monthly payment.

' Put APR in proper form.
IF APR > 1 THEN APR = APR / 100

Payment = ABS(-Pmt#(APR / YEARMONTHS, NumPeriods, PresentVal, FutureVal,_
                    PaymentType, Status))

' Examine Status to determine success of failure of any financial function.
IF Status THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the monthly payment."
ELSE
    ' Display the calculated monthly payment.
    PRINT "Your monthly payment is ";
    PRINT USING DOLLARFORMAT$; Payment
    PRINT
    PRINT "Would you like a breakdown of the principal and interest by month"
    PRINT "for the entire period of the loan? ";
    DO WHILE Answer$ <> "Y" AND Answer$ <> "N"
        Answer$ = UCASE$(INPUT$(1))
    LOOP
    PRINT Answer$
    PRINT
    IF Answer$ = "Y" THEN
        CLS
        PRINT "Month", "Payment", "Principal", "Interest"
        PRINT
        FOR Period = 1 TO NumPeriods
            ' Calculate the principal part of the payment.
            PrincPmt = ABS(-PPmt#(APR / YEARMONTHS, Period, NumPeriods,_
                           -PresentVal, FutureVal, PaymentType, Status))
            ' Round the result.
            PrincPmt = (INT((PrincPmt + .005) * 100) / 100)

            ' Calculate the interest part of the payment.
            ' IntPmt = -IPmt#(APR / YEARMONTHS, Period, NumPeriods,_
                           PresentVal, FutureVal, PaymentType, Status)
            IntPmt = Payment - PrincPmt
            ' Round the result.
            IntPmt = (INT((IntPmt + .005) * 100) / 100)
```

```
' Examine Status to determine success or failure of functions.
            IF Status% THEN
                ' If unsuccessful, announce a problem.
                PRINT "There was an error in calculating the interest or"
                PRINT " principal for period"; Period; "of your loan."
                END
            ELSE
                PRINT USING "###"; Period;
                PRINT TAB(15);
                PRINT USING DOLLARFORMAT$; Payment; PrincPmt; IntPmt
            END IF
        NEXT Period
    END IF
END IF
```

PPmt# Function

Action Returns payment on the principal for a given period of an annuity based on periodic, constant payments and a constant interest rate.

Syntax **PPmt#** (*rate#*, *per#*, *nper#*, *pv#*, *fv#*, *type%*, *status%*)

Remarks An annuity is a series of constant cash payments made over a continuous period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **PPmt#** function uses the following arguments:

Argument	Description
rate#	The interest rate per period. For example, if you get a car loan at an annual rate of 10 percent and make monthly payments, the rate per period would be .10 divided by 12, or .0083.
per#	The payment period; must be between 1 and *nper#*, inclusive. If it is not in that range, BASIC generates the error message `Overflow`.
nper#	The number of payment periods in the annuity. For example, if you get a four-year car loan and make monthly payments, your loan has a total number of 4 times 12, or 48, payment periods.
pv#	The present value or a lump sum that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv#	The future value or the cash balance you want to be attained sometime in the future after the last payment is made. The future value of a loan, for example is zero. As another example, if you think you will need $50,000 in 18 years to pay for your child's education, the future value is $50,000.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type%* if payments are due at the end of the period; use 1 if due at the beginning of the period.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *rate#* and *nper#* must use consistent units. For example:

rate#	*nper#*	**Loan description**
.10/12	4*12	Monthly payment, four-year loan, 10 percent annual interest
.10	4	Annual payment, four-year loan, 10 percent annual interest

Note For all arguments, cash you pay out, such as deposits to savings, is represented by negative numbers; cash you receive, such as dividend checks, is represented by positive numbers.

A number of other functions are related to **PPmt#**:

- **FV#** returns the the future value of an annuity.
- **IPmt#** returns the interest payment for an annuity for a given period.
- **NPer#** returns the number of periods (payments) for an annuity.
- **Pmt#** returns the periodic total payment for an annuity.
- **PV#** returns the present value of an annuity.
- **Rate#** returns the interest rate per period of an annuity.

To use **PPmt#** in the QBX environment, use the FINANCER.QLB Quick library. To use **PPmt#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	*Compiler options*
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also

FV#, IPmt#, NPer#, Pmt#, PV#, Rate#

Example

See the **Pmt#** function programming example, which uses the **PPmt#** function.

PV# Function

Action

Returns the present value, or lump sum, that a series of payments to be paid in the future is worth now.

Syntax

PV# (*rate#*, *nper#*, *pmt#*, *fv#*, *type%*, *status%*)

Remarks

The **PV#** function uses the following arguments:

Argument	Description
rate#	The interest rate per period. For example, if you get a car loan at an annual rate of 10 percent and make monthly payments, the rate per period would be .10 divided by 12, or .0083.
nper#	The number of payment periods in the annuity. For example, if you get a four-year car loan and make monthly payments, your loan has a total number of 4 times 12, or 48, payment periods.
pmt#	The payment to be made each period. It usually contains principal and interest and does not change over the life of the annuity.
fv#	The future value or the cash balance you want to be attained sometime in the future after the last payment is made. The future value of a loan, for example is zero. As another example, if you think you will need $50,000 in 18 years to pay for your child's education, the future value is $50,000.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type%* if payments are due at the end of the period; use 1 if due at the beginning of the period.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *rate#* and *nper#* must use consistent units. For example:

rate#	***nper#***	**Loan description**
.10/12	4*12	Monthly payment, four-year loan, 10 percent annual interest
.10	4	Annual payment, four-year loan, 10 percent annual interest

Note

For all arguments, cash you pay out, such as deposits to savings, is represented by negative numbers; cash you receive, such as dividend checks, is represented by positive numbers.

PV# uses the formula:

$$pv*(1+rate)^{nper}+pmt(1+rate*type)*\left(\frac{(1+rate)^{nper}-1}{rate}\right)+fv=0$$

Two other functions are related to **PV#**:

- **IPmt#** returns the interest payment for an annuity for a given period.
- **PPmt#** returns the principal payment for an annuity for a given period.

To use **PV#** in the QBX environment, use the FINANCER.QLB Quick library. To use **PV#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **IPmt#, PPmt#**

Example The following example uses **PV#** to return the present value of an annuity.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files also must be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST ENDPERIOD = 0
CONST BEGINPERIOD = 1
CONST DOLLARFORMAT$ = "$#,###,##0.00"
DEFDBL A-Z
DIM APR AS SINGLE
DIM Status AS INTEGER
APR = .102
Period = 20
YearlyIncome = 50000
FutureValue = 1000000

' Explain the premise.
CLS
PRINT "It seems you've won a million dollar lottery and you're going to"
PRINT "get $50,000.00 each year for the next 20 years."
PRINT
PRINT "You're curious about how much it's actually costing the people who"
PRINT "run the lottery to pay you that $1,000,000.00 over 20 years."
PRINT
PRINT "Assume that the prevailing interest rate is 10.2% compounded annually."
PRINT

' Calculate the present value of an annuity.
PresentValue = PV#(APR, 20, -YearlyIncome, FutureValue, BEGINPERIOD, Status)

' Examine Status to determine success of failure of PV#.
IF Status THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the present value of an annuity."
ELSE
    ' If successful, display the results.
    PresentValue$ = FormatD$(PresentValue, DOLLARFORMAT$)
    PRINT "Calculations show that the lottery people only have to deposit ";
    PRINT PresentValue$; ""
    PRINT "to ensure that you get your money."
END IF
```

Rate# Function

Action Returns the interest rate per period for an annuity.

Syntax **Rate#** (*nper#, mt#, pv#, fv#, type%, guess#, status%*)

Remarks An annuity is a series of constant cash payments made over a continuous period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The **Rate#** function uses the following arguments:

Argument	Description
nper#	The number of payment periods in the annuity. For example, if you get a four-year car loan and make monthly payments, your loan has a total number of 4 times 12, or 48, payment periods.
pmt#	The payment to be made each period. It usually contains principal and interest and does not change over the life of the annuity.
pv#	The present value or a lump sum that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.
fv#	The future value or the cash balance you want to be attained sometime in the future after the last payment is made. The future value of a loan, for example is zero. As another example, if you think you will need $50,000 in 18 years to pay for your child's education, the future value is $50,000.
type%	An integer expression that indicates when payments are due. Use 0 as the value for *type%* if payments are due at the end of the period; use 1 if due at the beginning of the period.
guess#	A value you guess is close to the result of **Rate#**.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

Note For all arguments, cash you pay out, such as deposits to savings, is represented by negative numbers; cash you receive, such as dividend checks, is represented by positive numbers.

Rate# is calculated by iteration. Starting with the value of *guess#*, **Rate#** cycles through the calculation until the result is accurate within .00001 percent. If after 20 tries it can't find a result that works, **Rate#** returns a status of 1. In most cases you can assume the *guess#* argument to be 0.1 (10 percent). However, if **Rate#** returns a status of 1 (failure), or if the result is not close to what you expected, try different values for it.

A number of other functions are related to **Rate#**:

- **FV#** returns the the future value of an annuity.
- **IPmt#** returns the interest payment for an annuity for a given period.
- **NPer#** returns the number of periods (payments) for an annuity.
- **Pmt#** returns the periodic total payment for an annuity.
- **PPmt#** returns the principal payment for an annuity for a given period.
- **PV#** returns the present value of an annuity.

To use **Rate#** in the QBX environment, use the FINANCER.QLB Quick library. To use **Rate#** outside of the QBX environment, link your program with the appropriate FINANCxx.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	*Compiler options*
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **FV#, IPmt#, NPer#, Pmt#, PPmt#, PV#**

Example The following example uses the **Rate#** function to determine the interest rate of a loan.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files must also be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST ENDPERIOD = 0
CONST BEGINPERIOD = 1
CONST PERCENT$ = "#0.0"
DEFDBL A-Z
DIM Status AS INTEGER
PresentValue = 8000
Payment = 200
Guess = .5

' Explain the premise.
CLS
PRINT "Your brother-in-law says he'll loan you $8000.00 if you agree to"
PRINT "repay the loan over the next 4 years at $200.00 a month. What he"
PRINT "doesn't tell you is the interest rate he's charging. Banks will "
PRINT "charge you at least 12% a year. You need to know what your brother-"
PRINT "in-law's interest rate is so you can get the best deal."
PRINT

' Calculate the interest rate.
IntRate = (Rate#(48, -Payment, PresentValue, 0, BEGINPERIOD, Guess, Status) * 12)
IntRate = IntRate * 100

' Examine Status% to determine success of failure of RATE.
IF Status% THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating the interest rate."
ELSE
    ' If successful, format and display the results.
    IntRate$ = FormatD$(IntRate, PERCENT$)
    PRINT "Calculations show that your brother-in-law is only going to charge"
    PRINT "you "; IntRate$; "% per year. He sounds like a nice guy."
END IF
```

Second& Function

Action Takes a date/time serial number and returns the second.

Syntax **Second&** (*serial#*)

Remarks The **Second&** function returns an integer between 0 and 59, inclusive, that represents the second corresponding to the argument.

The argument serial# is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Second&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Second&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&**, **Hour&**, **Minute&**, **Month&**, **Now#**, Serial Numbers, **Weekday&**, **Year&**

Example See the **TimeSerial#** function programming example, which uses the **Second&** function.

Serial Numbers

A serial number is a date/time code used by BASIC to represent dates and times between Jan. 1, 1753 and Dec. 31, 2078.

Serial-Number Format

Numbers to the left of the decimal point represent the date; numbers to the right of the decimal point represent the time. For example:

Serial number	Date and time represented
20323.25	August 22, 1955 6:00 A.M.
20324.25	August 23, 1955 6:00 A.M.
367.5	January 1, 1901 12:00 P.M.
367.75	January 2, 1901 6:00 P.M.

A serial number with no fractional part represents a date only. A serial number with only a fractional part represents a time only.

The date portion of the serial number (to the left of the decimal point) can represent dates ranging from January 1, 1753, through December 31, 2078. Dates before December 30, 1899 are represented by negative numbers; dates after December 30, 1899 are represented by positive numbers:

Serial number	Date represented
–1	December 29, 1899
0	December 30, 1899
1	December 31, 1899
2	January 1, 1900

The time portion of the serial number (to the right of the decimal point) can represent times that range from 0 (12:00:00 A.M.) to .99999 (11:59:59 P.M. or 23:59:59).

SetFormatCC Routine

Action Sets the country code for the **Format*X*$** functions.

Syntax **SetFormatCC** (*countrycode%*)

Remarks The argument *countrycode%* is the international telephone dialing prefix for the country chosen as the target audience when using the **Format*X*$** functions.

The only reason to use **SetFormatCC** is to change what the **Format*X*$** functions expect as thousands-separator and decimal-point formatting characters.

The default setting for the country code is the United States (1). If you set the country code to France (33), the following statement would display the number 1,000 (U.S.) as 1.000,00:

```
PRINT FormatD$ (1000, "#.##0,00")
```

The format you use with the **Format*X*$** functions must be compatible with the current country code in order for formatted numbers to display properly.

If you set the country code to one of the following countries, the **Format*X*$** functions expect alternative formatting characters: Austria, Belgium, Brazil, Denmark, France, Germany, Italy, Netherlands, Norway, Spain, Sweden, or Switzerland. Canada uses one country code for French Canada (2) and another for the remainder of Canada (1).

To use **SetFormatCC** in the QBX environment, use the DTFMTER.QLB Quick library. To use **SetFormatCC** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The FORMAT.BI header file contains the necessary function declarations for using the formatting functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Format*X*$**; **StringAddress** Routine, **StringLength** Routine (in Part 1).

Example See the **CCUR** function programming example, which uses the **SetFormatCC** routine.

SLN# Function

Action Returns straight-line depreciation of an asset for a single period.

Syntax **SLN#** (*cost#*, *salvage#*, *life#*, *status%*)

Remarks The **SLN#** function uses the following arguments:

Argument	Description
cost#	The initial cost of the asset.
salvage#	The value at the end of the useful life of the asset.
life#	The useful life of the asset (or the number of periods over which the asset is being depreciated).
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

To use **SLN#** in the QBX environment, use the FINANCER.QLB Quick library. To use **SLN#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide.*

See Also **DDB#, SYD#**

Example The following example uses **SLN#** to return the straight-line depreciation of an asset for a single period of its useful life.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files also must be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST YEARMONTHS = 12
CONST DOLLARFORMAT$ = "$#,###,##0.00"
DEFDBL A-Z
DIM Status  AS INTEGER

' Get user input.
CLS
PRINT "Assume you have a piece of expensive manufacturing equipment."
INPUT "Enter the original cost of the equipment"; Cost
PRINT
PRINT "Salvage value is the value of the asset at the end of its useful life."
INPUT "What is the salvage value of the equipment"; SalvageVal
PRINT
INPUT "What is the useful life of the equipment in months"; Life
IF Life < YEARMONTHS THEN
    PRINT
    PRINT "If the life is less than a year, it is not an asset."
    INPUT "Re-enter the useful life of the equipment in months"; Life
END IF
YearsInLife = Life / YEARMONTHS
' Round up to a whole year.
IF YearsInLife <> INT(Life / YEARMONTHS) THEN
    YearsInLife = INT(YearsInLife + 1)
END IF
PRINT

' Calculate and format the results.
PeriodDepreciation = SLN#(Cost, SalvageVal, YearsInLife, Status)

' Examine Status to determine success of failure of SLN#.
IF Status THEN
    ' If unsuccessful, announce a problem.
    PRINT "There was an error in calculating depreciation."
ELSE
    ' If successful calculate, format, and display the results.
    TotalDepreciation = Cost - SalvageVal
    MonthDepreciation = PeriodDepreciation / YEARMONTHS
```

```
    Cost$ = FormatD$(Cost, DOLLARFORMAT$)
    SalvageVal$ = FormatD$(SalvageVal, DOLLARFORMAT$)
    TotalDepreciation$ = FormatD$(TotalDepreciation, DOLLARFORMAT$)
    PeriodDepreciation$ = FormatD$(PeriodDepreciation, DOLLARFORMAT$)
    MonthDepreciation$ = FormatD$(MonthDepreciation, DOLLARFORMAT$)

    PRINT "If the original cost of your equipment is "; Cost$; ","
    PRINT "and the salvage value of that equipment is "; SalvageVal$; ","
    PRINT "the total depreciation at the end of its"; Life; "month lifetime "
    PRINT "is "; TotalDepreciation$; ". "
    PRINT
    PRINT "The straight-line depreciation is "; PeriodDepreciation$; " per year"
    PRINT "or "; MonthDepreciation$; " per month."
END IF
```

SYD# Function

Action Returns the sum-of-years' digits depreciation of an asset for a specified period.

Syntax **SYD#** (*cost#*, *salvage#*, *life#*, *period#*, *status%*)

Remarks The **SYD#** function uses the following arguments:

Argument	Description
cost#	The initial cost of the asset.
salvage#	The value at the end of the useful life of the asset.
life#	The useful life of the asset (or the number of periods over which the asset is being depreciated).
period#	Period for which asset depreciation is desired.
status%	A BASIC variable that indicates whether calculation succeeded or failed. The value of the variable will be 0 if the calculation was successful, and 1 if it was not.

The arguments *life#* and *period#* must use the same units. For example, if *life#* is given in months, *period#* also must be given in months.

To use **SYD#** in the QBX environment, use the FINANCER.QLB Quick library. To use **SYD#** outside of the QBX environment, link your program with the appropriate FINANC*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
FINANCER.LIB	80x87 or emulator math; DOS or OS/2 real mode
FINANCAR.LIB	Alternate math; DOS or OS/2 real mode
FINANCEP.LIB	80x87 or emulator math; OS/2 protected mode
FINANCAP.LIB	Alternate math; OS/2 protected mode

The FINANC.BI header file contains the necessary function declarations for using the financial functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **DDB#**, **SLN#**

Example The following example uses **SYD#** to return the depreciation of an asset for a specific period using the sum-of-years' digits method based on the asset's initial cost, salvage value, and the useful life of the asset.

To run this example, you must link it with the appropriate FINANC*xx*.LIB and DTFMT*xx*.LIB files or use the FINANCER.QLB and DTFMTER.QLB Quick libraries. The following include files must also be present:

```
' $INCLUDE: 'FINANC.BI'
' $INCLUDE: 'FORMAT.BI'
CONST YEARMONTHS = 12
CONST DOLLARFORMAT$ = "$#,###,##0.00"
DEFDBL A-Z
DIM Status  AS INTEGER

' Get user input.
CLS
PRINT "Assume you have a piece of expensive manufacturing equipment."
INPUT "Enter the original cost of the equipment"; Cost
PRINT
PRINT "Salvage value is the value of the asset at the end of its useful life."
INPUT "What is the salvage value of the equipment"; SalvageVal
PRINT
INPUT "What is the useful life of the equipment in months"; Life
IF Life < YEARMONTHS THEN
    PRINT
    PRINT "If the life is less than a year, it is not an asset."
    INPUT "Re-enter the useful life of the equipment in months"; Life
END IF
YearsInLife = Life / YEARMONTHS
' Round up to a whole year.
IF YearsInLife <> INT(Life / YEARMONTHS) THEN
    YearsInLife = INT(YearsInLife + 1)
END IF
PRINT
PRINT "For what year of the asset's life do you want to calculate ";
INPUT "depreciation "; DeprYear
DO WHILE Depr < YearsInLife AND DeprYear > YearsInLife
    PRINT
    PRINT "The year you enter must be at least 1, and not more than";
    PRINT YearsInLife
    INPUT "Reenter the year"; DeprYear
LOOP
PRINT
```

```
' Calculate and format the results.
PeriodDepreciation = SYD#(Cost, SalvageVal, YearsInLife, DeprYear, Status)

' Examine Status to determine success of failure of SYD#.
IF Status THEN
    'If unsuccessful, announce a problem.
    PRINT "There was an error in calculating depreciation."
ELSE
    ' If successful, calculate, format, and display the results.
    TotalDepreciation = Cost - SalvageVal
    MonthDepreciation = PeriodDepreciation / YEARMONTHS

    Cost$ = FormatD$(Cost, DOLLARFORMAT$)
    SalvageVal$ = FormtD$(SalvageVal, DOLLARFORMAT$)
    TotalDepreciation$ = FormatD$(TotalDepreciation, DOLLARFORMAT$)
    PeriodDepreciation$ = FormatD$(PeriodDepreciation, DOLLARFORMAT$)
    MonthDepreciation$ = FormatD$(MonthDepreciation, DOLLARFORMAT$)

    PRINT "If the original cost of your equipment is "; Cost$; ","
    PRINT "and the salvage value of that equipment is "; SalvageVal$; ","
    PRINT "the total depreciation at the end of its"; Life; "month lifetime "
    PRINT "is "; TotalDepreciation$; ". ";
    PRINT "The depreciation in year"; DeprYear
    PRINT "is "; PeriodDepreciation$; " or "; MonthDepreciation$; " per month."
END IF
```

TimeSerial# Function

Action Returns a serial number that represents the time of the arguments.

Syntax **TimeSerial#** (*hour%*, *minute%*, *second%*)

Remarks The **TimeSerial#** function uses the following arguments:

Argument	Description
hour%	An hour between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive.
minute%	A minute between 0 and 59, inclusive.
second%	A second between 0 and 59, inclusive.

You can use negative numbers as arguments, as long as the resulting serial number is positive. For example, to find the serial number for the time 50 seconds before 2:00:02, you could use:

```
TimeSerial#(2,0,2-50)
```

For more information, see the topic "Serial Numbers" in this section.

To use **TimeSerial#** in the QBX environment, use the DTFMTER.QLB Quick library. To use **TimeSerial#** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see "Creating and Using Quick Libraries" and "Using LINK and LIB" in the *Programmer's Guide*.

See Also **Hour&**, **Minute&**, **Now#**, **Second&**, Serial Numbers, **TimeValue#**

Example

The following example calculates the time between now and midnight tonight. The results of the calculations are shown as a total number of seconds, as a combination of hours, minutes, and seconds, and also as a standard time format. The time information is displayed using the **TimeValue#**, **TimeSerial#**, **Hour&**, **Minute&**, and **Second&** functions.

To run this example, you must link it with the appropriate DTFMT*xx*.LIB file or use the DTFMTER.QLB Quick library. The following include files must also be present.

```
' $INCLUDE: 'DATIM.BI'
' $INCLUDE: 'FORMAT.BI'

' Get a time value for midnight tonight -1 second.
Midnight# = TimeValue#("23:59:59")

' Get a time value for right now.
Instant# = Now#

' Get the difference in hours, minutes, seconds.
HourDiff% = Hour&(Midnight#) - Hour&(Instant#)
MinuteDiff% = Minute&(Midnight#) - Minute&(Instant#)

' Add in the odd second between 23:59:59 and midnight.
SecondDiff% = Second&(Midnight#) - Second&(Instant#) + 1
' Adjust so that seconds and minutes are less than 60.
IF SecondDiff% = 60 THEN
   MinuteDiff% = MinuteDiff% + 1
   SecondDiff% = 0
END IF

IF MinuteDiff% = 60 THEN
   HourDiff% = HourDiff% + 1
   MinuteDiff% = 0
END IF

' Calculate seconds between now and midnight.
TotalMinDiff# = (HourDiff% * 60) + MinuteDiff%
TotalSecDiff# = (TotalMinDiff# * 60) + SecondDiff%
' Put the difference back into a standard time format.
TotalDiff# = TimeSerial#(HourDiff%, MinuteDiff%, SecondDiff%)
' Display results.
CLS
PRINT "There are a total of"; TotalSecDiff#; "seconds until midnight."
PRINT "That translates to"; HourDiff%; "hours,"; MinuteDiff%;
PRINT "minutes, and"; SecondDiff%; "seconds."
PRINT
PRINT "The difference can also be expressed in standard time notation:"
PRINT FormatD$(TotalDiff#, "hh:mm:ss")
```

TimeValue# Function

Action Returns a serial number that represents the time of the argument.

Syntax **TimeValue#** (*time$*)

Remarks The argument *time$* is a time between 0:00:00 (12:00:00 A.M.) and 23:59:59 (11:59:59 P.M.), inclusive. Time can be entered as "2:24PM" or "14:24".

Any date information used in *time$* is ignored.

For more information, see the topic "Serial Numbers" in this section.

To use **TimeValue#** in the QBX environment, use the DTFMTER.QLB Quick library. To use **TimeValue#** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Hour&**, **Minute&**, **Now#**, **Second&**, Serial Numbers, **TimeSerial#**, **Year&**

Example See the **TimeSerial#** function programming example, which uses the **TimeValue#** function.

Weekday& Function

Action Takes a date-time serial number and returns the day of the week.

Syntax **Weekday&** (*serial#*)

Remarks The **Weekday&** function returns an integer between 1 (Sunday) and 7 (Saturday) that represents the day of the week corresponding to the argument.

The argument *serial#* is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Weekday&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Weekday&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	*Compiler options*
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&, Hour&, Minute&, Month&, Now#, Second&, Year&**

Example See the **DateValue#** function programming example, which uses the **Weekday&** function.

Year& Function

Action Takes a date-time serial number and returns the year.

Syntax **Year&** (*serial#*)

Remarks The **Year&** function returns an integer between 1753 and 2078, inclusive, that represents the year corresponding to the argument.

The argument *serial#* is a serial number that represents a date and/or time. For more information, see the topic "Serial Numbers" in this section.

To use **Year&** in the QBX environment, use the DTFMTER.QLB Quick library. To use **Year&** outside of the QBX environment, link your program with the appropriate DTFMT*xx*.LIB file. Depending on the compiler options you chose when you installed BASIC, one or more of the following files will be available:

Filename	Compiler options
DTFMTER.LIB	80x87 or emulator math; DOS or OS/2 real mode
DTFMTAR.LIB	Alternate math; DOS or OS/2 real mode
DTFMTEP.LIB	80x87 or emulator math; OS/2 protected mode
DTFMTAP.LIB	Alternate math; OS/2 protected mode

The DATIM.BI header file contains the necessary function declarations for using the date/time functions. For more information on using libraries, see Chapter 18, "Using LINK and LIB" and Chapter 19, "Creating and Using Quick Libraries" in the *Programmer's Guide*.

See Also **Day&**, **Hour&**, **Minute&**, **Month&**, **Now#**, **Second&**, Serial Numbers

Example See the **DateValue#** function programming example, which uses the **Year&** function.

Part 3

BASIC Toolbox Reference

Part 3 contains information and instructions for using the procedures in the Matrix Math, Presentation Graphics, and User Interface toolboxes.

First in Part 3 are summary tables that list tasks supported by each toolbox, the procedures to use, and the actions that result.

Next is overview and reference information for the matrix math functions contained in the Matrix Math toolbox.

The next section contains reference information for the procedures in the Presentation Graphics toolbox.

The last section of Part 3 contains an overview of the parts of the User Interface toolbox, instructions on how to use user interface procedures in your programs, and reference information for each procedure.

Toolbox Summary Tables

Each table in this section summarizes a group of procedures that are supported by toolbox files. Each table lists the procedure name, the task it performs, and any value returned if it is a **FUNCTION** procedure.

The following groups are summarized here:

- Matrix Math **FUNCTION** procedures
- Presentation Graphics **SUB** and **FUNCTION** procedures
- Font **SUB** and **FUNCTION** procedures
- User Interface **SUB** and **FUNCTION** procedures
  - Menu
  - Window
  - Mouse
  - General

Matrix Math FUNCTION Procedures

Table 3.1 summarizes the matrix math **FUNCTION** procedures that are provided by the MATB.BAS sample code module. There is a separate toolbox procedure for each supported numeric data type. The last letter in the procedure name identifies the data type of the procedure according to the following conventions:

- I is integer.
- L is long integer.
- S is single-precision floating point.
- D is double-precision floating point.
- C is currency.

Table 3.1 lists the type of operation performed by the matrix math procedures, the names of the procedures, and the results. The return value of each procedure is a result code that indicates either successful completion of the operation, or the reason the operation could not be completed. For more information about the result codes returned, see the detailed description of each of the procedures in "Matrix Math Toolbox," later in Part 3.

Table 3.1 Summary of Matrix Math FUNCTION Procedures

Operation	FUNCTION name	Result
Addition	**MatAddI%** **MatAddL%** **MatAddS%** **MatAddD%** **MatAddC%**	Finds the sum of two matrixes whose row and column dimensions are identical.
Subtraction	**MatSubI%** **MatSubL%** **MatSubS%** **MatSubD%** **MatSubC%**	Finds the difference between two matrixes whose row and column dimensions are identical.
Multiplication	**MatMultI%** **MatMultL%** **MatMultS%** **MatMultD%** **MatMultC%**	Finds the product of two matrixes where the first matrix has the same number of columns as the second matrix has rows.
Determinant	**MatDetI%** **MatDetL%** **MatDetS%** **MatDetD%** **MatDetC%**	Finds the determinant of a square matrix.
Inverse	**MatInvS%** **MatInvD%** **MatInvC%**	Finds the multiplicative inverse of a square matrix, if one exists; that is, if the determinant is not equal to 0.
Gaussian elimination	**MatSEqnS%** **MatSEqnD%** **MatSEqnC%**	Uses Gaussian elimination to solve, if a solution exists, a system of linear equations contained in a square matrix and a vector.

Presentation Graphics SUB and FUNCTION Procedures

Table 3.2 lists the procedures used in BASIC for making charts. The sample code module that supports these procedures is called CHRTB.BAS.

Table 3.2 Summary of Presentation Graphics SUB and FUNCTION Procedures

Procedure	Operation
Chart	Draws a bar, column, or line chart.
ChartMS	Draws a multi-series bar, column, or line chart.
ChartPie	Draws a pie chart.
ChartScatter	Draws a single-series scatter chart.
ChartScatterMS	Draws a multi-series scatter chart.
ChartScreen	Sets the screen mode to be used when displaying a chart.
DefaultChart	Initializes a `ChartEnvironment` structure for the specified chart type and style.
GetPaletteDef	Gets a copy of the current internal chart palette.
GetPattern$	Returns a string that can be used by BASIC as a pixel pattern
LabelChartH	Prints a user-defined string horizontally on a chart.
LabelChartV	Prints a user-defined string vertically on a chart.
MakeChartPattern$	Changes fill pattern and color.
ResetPaletteDef	Builds the internal chart palette for the current screen mode using the current style pool.
SetPaletteDef	Replaces the internal chart palette with user-defined values.

Font SUB and FUNCTION Procedures

Table 3.3 lists the procedures used in BASIC for working with fonts. The sample code module that supports these procedures is called FONTB.BAS.

Table 3.3 Summary of Font SUB and FUNCTION Procedures

Procedure	Operation
GetFontInfo	Gets font information for the currently selected font.
GetGTextLen%	Returns the pixel length of a string based on the currently selected font.
GetMaxFonts	Gets the maximum number of fonts that can be registered and loaded.

Table 3.3 *Continued*

Procedure	Operation
GetRFontInfo	Gets font information for the currently registered fonts.
GetTotalFonts	Gets the number of fonts currently registered and loaded.
LoadFont%	Loads the font information from the .FON files or memory for the specified fonts and returns the number of fonts actually loaded.
OutGText%	Outputs text in the currently selected font using the current graphics color at the current graphics cursor position. Returns pixel length of the character output.
RegisterFonts%	Registers font-header information from a specified .FON file and returns the number of fonts registered.
RegisterMemFont%	Registers the font-header information for fonts that reside in memory and returns the number of fonts registered.
SelectFont	Designates a loaded font as the active font.
SetGCharSet	Sets the character set used in subsequent graphics characters.
SetGTextColor	Sets the character color used in subsequent graphic characters.
SetGTextDir	Sets the horizontal or vertical orientation of graphics characters.
SetMaxFonts	Sets the maximum number of fonts that are allowed to be registered and loaded.
UnRegisterFonts	Removes registered fonts from memory.

User Interface Procedures

The following tables summarize the procedures that make up the User Interface toolbox. These functions are provided in four separate toolbox files. The files are:

- MENU.BAS
- WINDOW.BAS
- MOUSE.BAS
- GENERAL.BAS

Each table lists the type of operation performed by the procedure, the procedure name, and the result of the procedure. See the detailed description of each of the procedures later in this part for more information about how to use them in your programs.

Menu SUB and FUNCTION Procedures

Table 3.4 lists the procedures used in BASIC for implementing pull-down menus and quick keys in your programs. The sample code file that supports these procedures is called MENU.BAS.

Table 3.4 Summary of Menu SUB and FUNCTION Procedures

Operation	Procedure name	Result
Initialize variables and preprocess menus	**MenuInit**	Initializes global menu arrays and mouse driver servicing routines.
	MenuPreProcess	Performs calculations and builds indexes so menus run faster.
Define and display menus	**MenuColor**	Assigns color to various components of the menus.
	MenuSet	Defines the structure of menus and the quick keys associated with each menu selection.
	MenuShow	Draws the menu across the top line of the screen.
Change the state of menus or menu items	**MenuItemToggle**	Toggles the state of a menu item between enabled but not selected and enabled and selected.
	MenuSetState	Explicitly assigns the state of a menu item. States are: empty (menu item does not appear on the menu); enabled but not selected; enabled and selected; and disabled.
Process menu events	**MenuCheck**	Provides a numeric indication of what menu selection, if any, was made following a menu or shortcut-key event.
	MenuEvent$	Monitors user input to determine if a menu choice is being made with either the mouse or the keyboard.
	MenuInkey$	Performs a BASIC **INKEY$** function as well as a **MenuEvent$** and **ShortCutKeyEvent$** function.
	MenuOff	Turns off menu and shortcut-key event processing.
	MenuOn	Turns on menu and shortcut-key event processing that was previously turned off.

Table 3.4 *Continued*

Operation	Procedure name	Result
Shortcut keys	**ShortCutKeyDelete$**	Revokes any previous shortcut-key definitions associated with a particular menu item.
	ShortCutKeyEvent$	Polls user input to determine if a menu item was selected using a shortcut key.
	ShortCutKeySet	Assign shortcut keys to individual menu items.

Window SUB and FUNCTION Procedures

Table 3.5 lists the procedures used in BASIC for implementing alert boxes, edit fields, list boxes, and windows in your programs. The sample code file that supports these procedures is called WINDOW.BAS.

Table 3.5 Summary of Window SUB and FUNCTION Procedures

Operation	Procedure name	Result
Initialize window variables	**WindowInit**	Initializes global variables for all procedures in WINDOW.BAS.
Define windows	**WindowBox**	Draws a box with a single-line border at a designated position within the current window.
	WindowClose	Closes a specified window.
	WindowColor	Changes color characteristics of the current window.
	WindowOpen	Defines the size, position, color, and other characteristics of individual windows.
	WindowSetCurrent	Makes the specified window the current window.
Get information about windows	**WindowCols**	Returns the number of columns in the current window.
	WindowCurrent	Returns the number of the current window handle.
	WindowNext	Returns the number of the next available window handle.
	WindowRows	Returns the number of rows in the current window.

Table 3.5 *Continued*

Operation	Procedure name	Result
Display text in windows	**WindowCls**	Clears all text from within a window.
	WindowLine	Draws a horizontal line across a window at the row specified.
	WindowLocate	Sets the row and column of the window text cursor to define the next position where a character will be printed.
	WindowPrint	Prints text in a window at the position established by **WindowLocate**.
	WindowScroll	Scrolls text, in the current window, the number of lines specified.
Button actions	**ButtonClose**	Erases the specified button from the current window and deletes it from the global arrays.
	ButtonInquire	Returns a value that indicates the state of the specified button. Only used with button types 1, 2, 3, 6, and 7.
	ButtonOpen	Opens a button of a specified type and places it in the current window at the specified window coordinates.
	ButtonSetState	Sets the state of the specified button in the current window.
	ButtonToggle	Toggles the state of the specified button between selected and not selected.
Edit field actions	**EditFieldClose**	Erases the specified edit field from the current window and deletes it from the global arrays.
	EditFieldInquire	Returns the string associated with the specified edit field.
	EditFieldOpen	Opens an edit field and places it in the current window at specified coordinates.

Table 3.5 *Continued*

Operation	Procedure name	Result
Miscellaneous actions	**Alert**	Displays a window containing between one and three buttons that indicate user choices. Returns a value that indicates the number of the button selected by the user.
	Dialog	Returns a value that indicates what type of button, edit field, or window event occurred during window-event processing.
	ListBox	Displays a window containing a list box, a scroll bar, an OK button, and a Cancel button. Returns a value that indicates the number of the item selected in the list box, or 0 if Cancel is selected.
	MaxScrollLength	Returns a value that indicates how many positions are available on a specified scroll bar.

Mouse SUB Procedures

Table 3.6 lists the procedures used in BASIC for supporting a Microsoft or 100-percent compatible mouse in your programs. The sample code file that supports these procedures is called MOUSE.BAS.

Table 3.6 Summary of Mouse SUB Procedures

Operation	Procedure name	Result
Initialize the mouse	**MouseDriver**	Calls Interrupt 51 (33H) and passes parameters to the proper CPU registers.
	MouseInit	Initializes mouse driver servicing routines.
Establish limits of mouse movement on the screen	**MouseBorder**	Defines the area where the mouse can be used.
Manipulate the mouse	**MouseHide**	Turns off display of the mouse cursor.
	MousePoll	Polls the mouse driver to establish the position of the mouse cursor and the status of mouse buttons.
	MouseShow	Turns on display of the mouse cursor.

General SUB and FUNCTION Procedures

Table 3.7 lists the procedures used in BASIC for other miscellaneous operations in the User Interface toolbox. The sample code file that supports these procedures is called GENERAL.BAS.

Table 3.7 Summary of General SUB and FUNCTION Procedures

Operation	Procedure name	Result
Accept and evaluate keyboard input	**AltToASCII$**	Decodes extended key codes associated with the Alt key and returns an individual ASCII character.
	GetShiftState	Returns the status of one of eight possible shift states.
Manipulate screen areas	**AttrBox**	Changes color attributes of text within an area described by specified row and column coordinates.
	Box	Draws a box around a defined area using specified foreground and background colors.
	GetBackground	Saves an area of the screen into a named buffer.
	PutBackground	Restores to specified coordinates a previously saved screen area from a named buffer.
	Scroll	Scrolls the defined area a specified number of lines.

Matrix Math Toolbox

This section describes the six **FUNCTION** procedures supported by the Matrix Math toolbox:

- **MatAdd FUNCTION**
- **MatSub FUNCTION**
- **MatMult FUNCTION**
- **MatDet FUNCTION**
- **MatInv FUNCTION**
- **MatSEqn FUNCTION**

Matrix Math is the name of a BASIC toolbox (MATB.BAS) included with Microsoft BASIC. The toolbox contains a set of matrix-manipulation procedures that perform math operations on two-dimensional matrixes. There is a separate toolbox procedure for each supported numeric data type. Which one to use depends on the data type of the elements in the matrix. The last letter in the procedure name identifies the data type of the procedure, as follows:

Letter	*Numeric data type*
I	Integer
S	Single precision floating point
C	Currency
L	Long integer
D	Double precision floating point

All numeric data types are supported by **MatAdd**, **MatSub**, **MatMult**, and **MatDet**. Integer and long integer data types are not supported by **MatInv** and **MatSEqn**.

Each procedure returns a result code that indicates the success of the operation, or, in the case of an error, the nature of the problem encountered. The following result codes are defined:

Result code	*Significance*
0	Normal completion. No error occurred.
−1	Determinant for matrix is 0. No inverse of the input matrix exists.
−2	Matrix not square; doesn't have same number of rows as columns.
−3	Inside dimension not the same. The number of columns in *matrix1* is not the same as the number of rows in *matrix2*.

–4	Matrixes not of the same dimension. That is, there are not the same number of rows in *matrix1* that are in *matrix2* or there are not the same number of columns in *matrix1* that are in *matrix2*.
–5	Dimensions for the solution matrix not correctly declared. The matrix does not have the proper number of rows and columns.

Note By default all arrays are zero based; that is, the lower bound of an array is 0. You may wish to use `OPTION BASE 1` so that the numbers in the **DIM** statement accurately reflect the number of row and column elements in your matrixes.

Following is a description of the **FUNCTION** procedures in the Matrix Math toolbox.

MatAdd FUNCTION

MatAddI%
MatAddL%
MatAddS%
MatAddD%
MatAddC%

Syntax *errcode%* = **MatAdd***type%*(*matrix1*( ), *matrix2*( ))

Remarks **MatAdd***type%* performs matrix addition of two matrixes that are the same in both row (*m*) and column (*n*) dimensions. The input matrixes contain values whose data type is defined by the last letter in the procedure name (I, L, S, D, or C). The sum is returned in *matrix1*( ), replacing any previous values. The solution matrix contains values of the same data type as the input matrixes. Once the procedure has been performed, the contents of *matrix2*( ) are meaningless.

MatAdd*type%* uses the following arguments:

Argument	Description
matrix1( )	A matrix consisting of *m* x *n* dimensions.
matrix2( )	A matrix consisting of *m* x *n* dimensions.

A result code indicates the success or failure of the **FUNCTION** procedure. Possibilities are:

Result code	Significance
0	Normal completion. No error occurred.
–4	Matrixes not of the same dimension. That is, there are not the same number of rows in *matrix1* as there are in *matrix2* or there are not the same number of columns in *matrix1* as in *matrix2*.

MatSub FUNCTION

MatSubI%
MatSubL%
MatSubS%
MatSubD%
MatSubC%

Syntax *errcode%* = **MatSub***type%*(*matrix1*( ),*matrix2*( ))

Remarks The **MatSub***type%* procedure performs matrix subtraction of two matrixes whose row (*m*) and column (*n*) dimensions are identical. The input matrixes contain values whose data type is defined by the last letter in the procedure name (I, L, S, D, or C). The first matrix is subtracted from the second and the difference is returned in *matrix1*( ), replacing any previous values. The solution matrix contains values of the same data type as the input matrixes. Once the procedure has been performed, the contents of *matrix2*( ) are meaningless.

MatSub*type%* uses the following arguments:

Argument	Description
matrix1( )	A matrix consisting of *m* x *n* dimensions.
matrix2( )	A matrix consisting of *m* x *n* dimensions.

A result code is returned that indicates the success or failure of the **FUNCTION** procedure. Possible result codes are:

Result code	Significance
0	Normal completion. No error occurred.
–4	Matrixes not of the same dimension. That is, there are not the same number of rows in *matrix1* as there are in *matrix2* or there are not the same number of columns in *matrix1* as in *matrix2*.

MatMult FUNCTION

MatMultI%
MatMultL%
MatMultS%
MatMultD%
MatMultC%

Syntax *errcode%* = **MatMult***type%*(*matrix1*( ),*matrix2*( ),*matrix3*( ))

Remarks

The **MatMult***type%* procedure performs matrix multiplication of two matrixes. The first matrix must have the same number of columns (*n* in an *m* x *n* matrix) as the number of rows in the second matrix (*n* in an *n* x *k* matrix). The input matrixes contain values whose data type is defined by the last letter in the procedure name (I, L, S, D, or C). The first matrix is multiplied by the second and the product is returned in *matrix3*( ). The solution matrix contains values of the same data type as the input matrixes. Once the procedure has been performed, the contents of the input matrixes are meaningless.

MatMult*type%* uses the following arguments:

Argument	Description
matrix1( )	A matrix consisting of *m* x *n* dimensions.
matrix2( )	A matrix consisting of *n* x *k* dimensions.
matrix3( )	Solution matrix consisting of *m* x *k* dimensions.

A result code is returned that indicates the success or failure of the **FUNCTION** procedure. Possible result codes are:

Result code	Significance
0	Normal completion. No error occurred.
–3	Inside dimension not the same. The number of columns in *matrix1* is not the same as the number of rows in *matrix2*.
–5	Dimensions for the solution matrix not correctly declared. The matrix does not have the proper number of rows and columns.

Note

Matrix division is performed using the **MatInv***type%* procedure to multiply one matrix by the inverse of the second; that is, A/B = A * Inverse(B).

MatDet FUNCTION

MatDetI%
MatDetL%
MatDetS%
MatDetD%
MatDetC%

Syntax

errcode% = **MatDet***type%*(*matrix*( ), *determinant*)

Remarks

The **MatDet***type%* procedure finds the determinant of a square matrix; that is, a matrix that has the same number of rows and columns (*n* x *n*). The input matrix contains values whose data type is defined by the last letter in the procedure name (I, L, S, D, or C). The resulting determinant, of the same data type as the input matrix values, is placed in *determinant*. After the procedure is performed, the contents of *matrix*( ) are meaningless.

MatDet*type%* uses the following arguments:

Argument	Description
matrix( )	A matrix consisting of *n* x *n* dimensions.
determinant	The determinant for *matrix*( ).

A result code indicates the success or failure of the **FUNCTION** procedure. Possibilities are:

Result code	Significance
0	Normal completion. No error occurred.
−2	Matrix not square. The matrix does not have the same number of rows as columns.

MatInv FUNCTION

MatInvS%
MatInvD%
MatInvC%

Syntax

errcode% = **MatInv***type%*(*matrix*( ))

Remarks

The **MatInv***type%* procedure finds the multiplicative inverse of a square matrix; that is, a matrix that has the same number of rows and columns (*n* x *n*). The input matrix contains values whose data type is defined by the last letter in the procedure name (S, D, or C). The resulting inverse matrix, of the same data type as the input matrix values, is returned in *matrix*( ), replacing any previous values.

The argument *matrix*( ) is a matrix consisting of *n* x *n* dimensions.

A result code indicates the success or failure of the **FUNCTION** procedure. Possibilities are:

Result code	Significance
0	Normal completion. No error occurred.
−1	The determinant for the matrix is 0. No inverse of the input matrix exists.
−2	Matrix not square. The matrix does not have the same number of rows as columns.

Note Matrix division is performed using the **MatInv***type%* procedure to multiply one matrix by the inverse of the second, that is, A/B = A * Inverse(B).

MatSEqn FUNCTION

MatSEqnS%
MatSEqnD%
MatSEqnC%

Syntax *errcode%* = **MatSEqn***type%*(*matrix1*( ),*matrix2*( ))

Remarks The **MatSEqn***type%* procedure solves a system of linear equations contained in a square matrix; that is, a matrix that has the same number of rows and columns (*n* x *n*). Gaussian elimination is used to solve the equations. The input matrix contains values whose data type is defined by the last letter in the procedure name (S, D, or C). The first matrix is the square input matrix that contains the coefficients for a system of simultaneous equations. The second matrix, *matrix2*( ), is the space where the solution is returned. The solution space is an *n* x 1 matrix. The solution matrix contains values of the same data type as the input matrix. Once the procedure has been performed, the identity matrix is in *matrix1*( ).

MatSEqn*type%* uses the following arguments:

Argument	**Description**
matrix1( )	A matrix consisting of *n* x *n* dimensions.
matrix2( )	Solution array consisting of 1 x *n* dimensions.

A result code is returned that indicates the success or failure of the **FUNCTION** procedure. Possible result codes are:

Result code	**Significance**
0	Normal completion. No error occurred.
–1	Determinant for matrix is 0. No inverse of the input matrix exists.
–2	Matrix not square. The matrix does not have the same number of rows as columns.
–3	Inside dimension not the same. The number of columns in *matrix1* is not the same as the number of rows in *matrix2*.

Presentation Graphics Toolbox

The first part of this section describes the **SUB** and **FUNCTION** procedures that are supported by the Presentation Graphics toolbox. The second part describes the **SUB** and **FUNCTION** procedures that are supported by the Fonts toolbox. The Fonts toolbox can be used with the Presentation Graphics toolbox or independently to draw graphics text.

The Presentation Graphics toolbox (CHRTB.BAS), included with the BASIC 7.0 Compiler, contains a set of routines for defining, analyzing, and printing charts on the screen.

There are five kinds of charts, each available in two styles as shown here:

Style	Bar	Column	Line	Scatter	Pie
1	Plain	Plain	Lines and points	Lines and points	Percent displayed
2	Stacked	Stacked	Points only	Points only	No percent

To use the toolbox, you first must declare the dimensions for a variable as the user-defined type `ChartEnvironment`. The definition for this type is found in the CHRTB.BI header file. This file also contains definitions for constants that can be used as arguments to the procedures described in this section. Using these constants in place of hand-coded numerics will make your program cleaner and easier to debug.

Presentation Graphics Error Codes

If an error occurs during the execution of a presentation graphics procedure, the variable ChartErr will contain a non-zero value.

For non-BASIC errors, the error numbers can be tested using numerics or constants as defined in the CHRTB.BI header file. The meaning of a non-BASIC error is shown in Table 3.8.

Table 3.8 Presentation Graphics Error Codes

Number	Constant name	Type of error
15	cBadLogBase	LogBase <= 0
20	cBadScaleFactor	ScaleFactor = 0
25	cBadScreen	Invalid screen mode
30	cBadStyle	Invalid chart style
105	cBadDataWindow	Data window calculated too small
110	cBadLegendWindow	Legend-window coordinates invalid
135	cBadType	Invalid chart type

Table 3.8 *Continued*

Number	Constant name	Type of error
155	cTooFewSeries	Too few series (*first%* > *last%*)[1]
160	cTooSmallN	No data in series (*n%* = 0)[2]
165	cBadPalette	Palette not dimensioned correctly
170	cPalettesNotSet	**SetPaletteDef** procedure hasn't been used
175	cNoFontSpace	No more room for new fonts

1 See the **ChartMS** and **ChartScatterMS** procedures.
2 See the **Chart**, **ChartMS**, **ChartPie**, **ChartScatter**, and **ChartScatterMS** procedures.

Numbers greater than 100 are fatal errors and will cause charting routines to exit.

If BASIC generates an error, the value of ChartErr is equal to 200 plus the BASIC error number. (See Appendix D, "Error Messages" for a complete list of BASIC error messages.)

Chart SUB

Action Draws bar, column, and line charts.

Syntax **Chart** *env*, *cat$()*, *value!()*, *n%*

Remarks The **Chart** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
cat$()	A one-dimensional string array of category names.
value!()	A one-dimensional single-precision array that contains the chart data.
n%	An integer that contains the number of data items in *value!()*.

The *value!()* and *cat$()* arrays must have a lower bound of 1.

The **AnalyzeChart** routine uses the same arguments as **Chart. AnalyzeChart** analyzes and defines parameters in the chart environment based on the input data, but it does not print a chart on the screen.

ChartMS SUB

Action Draws multi-series bar, column, and line charts.

Syntax **ChartMS** *env*, *cat$()*, *value!()*, *n%*, *first%*, *last%*, *serieslabel$()*

Remarks The **ChartMS** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
cat$()	A one-dimensional string array of category labels.
value!()	A two-dimensional single-precision array of values that contains multiple series of data.
n%	An integer that contains the number of data items in each series to be charted.
first%	An integer that indicates the first series in array *value!()* to be charted.
last%	An integer that indicates the last series in array *value!()* to be charted.
serieslabel$()	A one-dimensional string array that contains labels for the different data series.

Dimensions for the *value!()* and *serieslabel$()* arrays are declared as follows:

```
DIM Val!( 1 to n%, first% to last%)
DIM serieslabel$( first% to last%)
```

To chart all series, set *first%* to 1 and *last%* to the last series number in array *value!()*. To chart several contiguous series, set *first%* to the lowest series number and *last%* to the highest series number desired.

An analysis routine called **AnalyzeChartMS** (included in the Presentation Graphics toolbox) uses the same arguments as **ChartMS**. **AnalyzeChartMS** analyzes and defines parameters in the chart environment based on the input data, but it does not print a chart on the screen.

ChartPie SUB

Action Draws bar, column, and line charts.

Syntax **ChartPie** *env*, *cat$()*, *value!()*, *expl%()*, *n%*

Remarks The **ChartPie** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
cat$()	A one-dimensional string array of category names.
value!()	A one-dimensional single-precision array that contains the chart data.
expl%()	A one-dimensional integer array containing flags that determine whether each element of the pie chart is exploded. If the value of an *expl%()* array element is nonzero, that slice of the pie chart is exploded. If the array element is zero, the slice is not exploded.
n%	An integer that contains the number of data items in *value!()*.

The *value!()*, *cat$()*, and *expl%()* arrays must have a lower bound of 1.

An analysis routine called **AnalyzePie** (included in the Presentation Graphics toolbox) uses the same arguments as **ChartPie**. **AnalyzePie** analyzes and defines parameters in the chart environment based on the input data, but it does not print a chart on the screen.

ChartScatter SUB

Action Draws single-series scatter charts.

Syntax **ChartScatter** *env*, *valx!()*, *valy!()*, *n%*

Remarks The **ChartScatter** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
valx!()	A one-dimensional single-precision array of values for the x axis.

valy!()	A one-dimensional single-precision array of values for the y axis.
n%	An integer that contains the number of data items to be charted.

The *valx!()* and *valy!()* arrays must have a lower bound of 1.

An analysis routine called **AnalyzeScatter** (included in the Presentation Graphics toolbox) uses the same arguments as **ChartScatter**. **AnalyzeScatter** analyzes and defines parameters in the chart environment based on the input data, but it does not print a chart on the screen.

ChartScatterMS SUB

Action Draws multi-series scatter charts.

Syntax **ChartScatterMS** *env*, *valx!()*, *valy!()*, *n%*, *first%*, *last%*, *serieslabel$()*

Remarks The **ChartScatterMS** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
valx!()	A two-dimensional single-precision array of values for multiple series of data for the x axis.
valy!()	A two-dimensional single-precision array of values for multiple series of data for the y axis.
n%	An integer that contains the number of data items to be charted.
first%	An integer that indicates the first series to be charted.
last%	An integer that indicates the last series to be charted.
serieslabel$()	A one-dimensional string array that contains labels for the different data series.

Dimensions for the *valx!()*, *valy!()* and *serieslabel$()* arrays are declared as follows:

```
DIM valx!( 1 to n%, first% to last%)
DIM valy!( 1 to n%, first% to last%)
DIM serieslabel$( first% to last%)
```

To chart all series, set *first%* to 1 and *last%* to the last series number in array *value!()*. To chart several contiguous series, set *first%* to lowest series number and *last%* to the highest series number desired.

An analysis routine called **AnalyzeScatterMS** (included in the Presentation Graphics toolbox) uses the same arguments as **ChartScatterMS**. **AnalyzeScatterMS** analyzes and defines parameters in the chart environment based on the input data, but it does not print a chart on the screen.

ChartScreen SUB

Action Sets the screen mode to be used when displaying a chart.

Syntax **ChartScreen** *n%*

Remarks The argument *n%* is an integer that contains a valid screen-mode number.

This procedure must be used to set the screen mode instead of the standard BASIC **SCREEN** statement.

If an invalid screen mode is used, the variable ChartErr will contain the number 25.

See Also **SCREEN** Statement

DefaultChart SUB

Action Initializes the elements of the variable type `ChartEnvironment` for the specified chart type and style.

Syntax **DefaultChart** *env*, *type%*, *style%*

Remarks The **DefaultChart** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment.`
type%	An integer that defines the type of chart (1–5).
style%	An integer that defines the style of the chart (1–2).

The variable *type%* can use either numeric data or constants found in the CHRTB.BI file to define the chart as shown in the following table:

Value	Constant	Type of chart
1	cBar	Bar
2	cColumn	Column
3	cLine	Line
4	cScatter	Scatter
5	cPie	Pie

The value of the variable *style%* determines the style of the chart as shown here:

Value	Bar	Column	Line	Scatter	Pie
1	Plain	Plain	Lines & points	Lines & points	Percent
2	Stacked	Stacked	Points only	Points only	No percent

For program clarity, the following constants (found in the CHRTB.BI file) should be used in place of numeric arguments:

Numeric argument	Equivalent constant
1	cPlain, cLines, cPercent
2	cStacked, cNoLines, cNoPercent

GetPaletteDef SUB

Action Gets a copy of the current internal chart palette.

Syntax **GetPaletteDef** *palc%(), pals%(), palp$(), palch%(), palb%()*

Remarks The **GetPaletteDef** procedure uses the following arguments:

Argument	Description
palc%()	A one-dimensional integer array of color numbers corresponding to the palette entries.
pals%()	A one-dimensional integer array of line styles corresponding to the palette entries.

palp$()	A one-dimensional string array of fill patterns corresponding to the palette entries.
palch%()	A one-dimensional integer array of plot-character numbers corresponding to the palette entries.
palb%()	A one-dimensional integer array of line styles used for drawing window borders and grid lines.

Dimensions for each array should be declared from 0 to cPalLen, a constant found in the CHRTB.BI header file. Note that all palette arrays have a lower bound of 0 (not 1, which is used for data and category-string arrays).

For more information on fill patterns, see the entries for **PAINT** and **PALETTE** in Part 1, "Language Reference."

GetPattern$ FUNCTION

Action Returns a string that can be used by BASIC as a pixel pattern.

Syntax **GetPattern$**(*bits%*, *patternNum%*)

Remarks The **GetPattern$** and **MakeChartPattern$** procedures are used in combination with the **GetPaletteDef** and **SetPaletteDef** procedures to change the fill pattern for pie, column, and bar charts. **GetPattern$** constructs a value for *refPattern$*, which is one of the arguments used by **MakeChartPattern$**.

The **GetPattern$** procedure uses the following arguments:

Parameter	Description
bits%	Use 2 for screen mode 1, 8 for screen mode 13, and 1 for all other screen modes.
patternNum%	An integer between one and `cPalLen`.

For more information, see the entry for the **MakeChartPattern$** procedure later in this section.

LabelChartH SUB

Action Prints a user-defined string horizontally on a chart.

Syntax **LabelChartH** *env*, *x%*, *y%*, *font%*, *color%*, *string$*

Remarks The **LabelChartH** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
x%	An integer that indicates the left position of the first character, in pixels, relative to the chart window.
y%	An integer that indicates the bottom of the first character, in pixels, relative to the chart window.
font%	An integer that contains the number of the font (in currently loaded list) to use.
color%	An integer that contains the color number in the chart palette used to assign color to the string.
string$	The text string to be printed.

The **LabelChartH** procedure must be called after calling the charting sub.

If an invalid font number (such as 0) is contained in *font%*, the first font loaded is used. If no fonts are loaded, the default font contained in the Presentation Graphics toolbox files is used.

LabelChartV SUB

Action Prints a user-defined string vertically on a chart.

Syntax **LabelChartV** *env*, *x%*, *y%*, *font%*, *color%*, *string$*

Remarks The **LabelChartV** procedure uses the following arguments:

Argument	Description
env	A variable dimensioned as type `ChartEnvironment`.
x%	An integer that indicates the left position of the first character, in pixels, relative to the chart window.

y%	An integer that indicates the top of the first character, in pixels, relative to the chart window.
font%	An integer that contains the number of the font (in currently loaded list) to use.
color%	An integer that contains the color number in the chart palette used to assign color to the string.
string$	The text string to be printed.

The **LabelChartV** procedure must be called after calling the charting sub.

Each character is printed vertically and appears in a vertical column.

If an invalid font number (such as 0) is contained in *font%*, the first font loaded is used. If no fonts are loaded, the default font contained in the Presentation Graphics toolbox files is used.

MakeChartPattern$ FUNCTION

Action Changes fill pattern and color.

Syntax **MakeChartPattern$**(*refPattern$*, *foreground%*, *background%*)

Remarks The **MakeChartPatern$** and **GetPattern$** procedures are used in combination with the **GetPaletteDef** and **SetPaletteDef** procedures to change the fill pattern for pie, column, and bar charts.

The **GetPattern$** procedure uses the following arguments:

Argument	Description
refPattern$	A string representing the pixel pattern.
foreground%	Attribute to map to the pixels in *refPattern$* that are defined as being on.
background%	Attribute to map to the pixels in *refPattern$* that are defined as off.

Note that if *foreground%* and *background%* are the same value, the fill pattern appears as a solid color.

Constructing a value for *refPattern$* is difficult. When using the Presentation Graphics toolbox, you can simplify the process by using the **GetPattern$** procedure to choose an internally-defined pattern. **MakeChartPattern$** maps colors to the string returned by **GetPattern$** to produce the combination of color and pattern you want. For more information, see the entry for the **GetPattern$** procedure earlier in this section.

ResetPaletteDef SUB

Action Rebuilds the internal chart palette for the current screen mode.

Syntax **ResetPaletteDef**

SetPaletteDef SUB

Action Replaces the internal chart palette with new values.

Syntax **SetPaletteDef** *palc%(), pals%(), palp$(), palch%(), palb%()*

Remarks The **SetPaletteDef** procedure uses the following arguments:

Argument	Description
palc%()	A one-dimensional integer array of color numbers corresponding to the palette entries.
pals%()	A one-dimensional integer array of line styles corresponding to the palette entries.
palp$()	A one-dimensional string array of fill patterns corresponding to the palette entries.
palch%()	A one-dimensional integer array of plot-character numbers corresponding to the palette entries.
palb%()	A one-dimensional integer array of line styles used for drawing window borders and grid lines.

The dimensions for each array must be declared from 0 to cPalLen, a constant found in the CHRTB.BI header file. Note that all palette arrays have a lower bound of 0 (not 1, which is used for data and category string arrays).

For more information on fill patterns, see the entries for **PAINT** and **PALETTE** in Part 1, "Language Reference."

Fonts Toolbox

The Fonts toolbox (FONTB.BAS), included with this version of BASIC, contains a set of routines that perform font-management tasks such as registering, loading, selecting, and printing fonts on the screen.

Two font files are supplied: HELVB.FON (Helvetica) and TMSRB.FON (Times Roman). Each of these files contains six font sizes. Besides these two font files, the Fonts toolbox will work with any standard Windows bitmap .FON file.

To use the Fonts toolbox, you must compile and link the FONTASM.OBJ and FONTB.BAS modules into a .LIB or .QLB library file. Once this library exists, you can use it just like any other library. The Fonts toolbox can be used independently of the Presentation Graphics toolbox to draw graphics text.

Several of the procedures use the font-header information defined in the user-defined type `FontInfo`. This type is defined in the FONTB.BI header file. This file also contains definitions for constants that can be used as arguments to the procedures described in this section. Using these constants in place of hand-coded numbers makes your program cleaner and easier to debug.

Font Error Codes

If an error occurs during the execution of a font **FUNCTION** or **SUB** procedure, the variable `FontErr` will contain a non-zero value.

For non-BASIC errors, the error numbers can be tested using numerics or constants as defined in the FONTB.BI header file. The meaning of a non-BASIC error is shown in Table 3.9.

Table 3.9 Font Error Codes

Number	Constant name	Type of error
1	cFileNotFound	Specified file not found
2	cBadFontSpec	Invalid part of a font specification
5	cBadFontFile	Invalid font-file format
6	cBadFontLimit	Invalid font limit
7	cTooManyFonts	Tried to exceed limit
8	cNoFonts	No loaded fonts
10	cBadFontType	Not a bitmap font
11	cBadFontNumber	Bad font number

Table 3.9 *Continued*

Number	Constant name	Type of error
12	cNoFontMem	Not enough memory
200	cFLUnexpectedOff	Unexpected BASIC error offset

If BASIC generates an error, the `FontErr` variable is set to cFLUnexpectedOff plus the value of the BASIC error.

GetFontInfo SUB

Action Gets font information about the currently selected font.

Syntax **GetFontInfo** (*fontinfo*)

Remarks The argument *fontinfo* is equivalent to the data structure type `FontInfo` and is used to receive the font-header information. The `FontInfo` type is defined as follows:

```
FontInfo
   TYPE FontInfo
      FontNum   AS  INTEGER
      Ascent    AS  INTEGER
      Points    AS  INTEGER
      PixWidth  AS  INTEGER
      PixHeight AS  INTEGER
      Leading   AS  INTEGER
      AvgWidth  AS  INTEGER
      MaxWidth  AS  INTEGER
      FileName  AS  STRING   *   cMaxFileName
      FaceName  AS  STRING   *   cMaxFaceName
   END TYPE
```

The following list describes the parts of the `FontInfo` data structure:

Part	Description
`FontNum`	The location of the font in the registered list or the loaded list.
`Ascent`	The pixel distance from the character baseline to the top of the character box.
`Points`	The point size of the current font as defined in the **LoadFont%** function.

`PixWidth`	The width of the character bitmap. A value of 0 specifies a proportional font. A nonzero value specifies the pixel width of the characters in a fixed-space font.
`PixHeight`	The height of the character bitmap.
`Leading`	The number of blank lines at the top of the character definition to act as leading between lines.
`AvgWidth`	The average width of characters in pixels.
`MaxWidth`	The pixel width of the widest character in the font.
`FileName`	The filename from which the font was loaded. The filename has the extension .FON.
`FaceName`	The name of the typeface (for example, Helvetica, Courier) taken from the filename.

Note

When using the **GetFontInfo** procedure, be sure you have access to the .FON file on disk so that it can find the `FontNum`, `FileName`, and `FaceName` values for the currently selected font.

For more information on the elements of the variable type `FontInfo`, see documentation on font file format in Chapter 7, "File Formats" in the *Programmer's Reference*, which is part of the Microsoft Windows Software Development Kit.

GetGTextLen% FUNCTION

Action

Returns the pixel length of a string based on the currently selected font.

Syntax

GetGTextLen% *(txt$)*

Remarks

GetGTextLen% is an integer that contains the pixel length of the string supplied in the argument *txt$*. The pixel length returned from the **GetGTextLen%** function is based on the currently selected font. From this pixel length, you then can determine whether a string will fit on a particular screen (for example, CGA, EGA).

GetMaxFonts SUB

Action Gets the maximum number of fonts that can be registered and loaded.

Syntax **GetMaxFonts** (*maxregistered%*, *maxloaded%*)

Remarks The **GetMaxFonts** procedure uses the following arguments:

Argument	Description
maxregistered%	An integer that is the maximum number of fonts that can be registered.
maxloaded%	An integer that is the maximum number of fonts that can be loaded.

GetMaxFonts returns the current number of fonts that can be registered and loaded as set in the **SetMaxFonts** routine. The default is 10.

See Also **GetTotalFonts**, **SetMaxFonts**

GetRFontInfo SUB

Action Gets the font information on the currently registered font.

Syntax **GetRFontInfo** (*n%*, *fontinfo*)

Remarks The **GetRFontInfo** procedure uses the following arguments:

Argument	Description
n%	The number of the registered font.
fontinfo	A variable of type `FontInfo` that receives the font-header information.

The `FontInfo` data structure type is defined as follows:

```
FontInfo
   TYPE FontInfo
      FontNum   AS INTEGER
      Ascent    AS INTEGER
      Points    AS INTEGER
      PixWidth  AS INTEGER
      PixHeight AS INTEGER
      Leading   AS INTEGER
      AvgWidth  AS INTEGER
      MaxWidth  AS INTEGER
      FileName  AS STRING   *   cMaxFileName
      FaceName  AS STRING   *   cMaxFaceName
   END TYPE
```

The following list describes the parts of the `FontInfo` data structure:

Part	Description
`FontNum`	The location of the font in the registered list or the loaded list.
`Ascent`	The pixel distance from the character baseline to the top of the character box.
`Points`	The point size of the current font as defined in the **LoadFont%** function.
`PixWidth`	The width of the character bitmap. A value of 0 specifies a proportional font. A non-zero value specifies the pixel width of the characters in a fixed-space font.
`PixHeight`	The height of the character bitmap.
`Leading`	The number of blank lines at the top of the character definition to act as leading between lines.
`AvgWidth`	The average width of characters in pixels.
`MaxWidth`	The pixel width of the widest character in the font.
`FileName`	The filename from which the font was loaded. The filename has the extension .FON.
`FaceName`	The name of the typeface (for example, Helvetica, Courier) taken from the filename.

Note When using **GetRFontInfo**, make sure you have access to the .FON file on disk so that it can find the `FontNum`, `FileName`, and `FaceName` values for the currently selected font.

For more information on the elements of the variable type `FontInfo`, see the Microsoft Windows developer documentation on font file format.

GetTotalFonts SUB

Action Gets the number of fonts currently registered and loaded.

Syntax **GetTotalFonts** (*registered%*, *loaded%*)

Remarks The **GetTotalFonts** procedure uses the following arguments:

Argument	Description
registered%	The number of currently registered fonts.
loaded%	The number of currently loaded fonts.

See Also **GetMaxFonts**, **SetMaxFonts**

LoadFont% FUNCTION

Action Loads the font information from the .FON files or memory for the specified registered fonts and returns the number of fonts actually loaded.

Syntax **LoadFont%** (*fontspec$*)

Remarks The argument *fontspec$* is a specification used to load one or more of the registered fonts. The specification for each font consists of one or more of the following:

Specification	Description
`n#`	Loads font number # in the list of currently registered fonts.
`t` *name*	Specifies the desired font *name*. (There must be a space between `t` and *name*.)
`s#`	Specifies point size #. When `s` is specified, only the fonts designed for the screen mode specified by `m` are used.
`h#`	Specifies pixel height # of a font.
`m#`	Specifies screen mode #. This is used for sizing fonts with `s`. The default is the current screen mode.
`b`	Selects the best fit based on the size specified in the `s` or `h` options. The specification `b` is ignored with vector fonts because they can be any size.

`f`	Loads only a single fixed-space font.
`p`	Loads only the first proportionally spaced font that registered.

Any previously loaded fonts are removed from memory. To load more than one font, multiple specifications are needed.

Multiple specifications are separated by slash characters, and blanks are ignored. If the specifications do not match a registered font or if an invalid font is specified, then font number one (`n1`) is used.

The **LoadFont%** function loads the character-bit maps into the loaded fonts array. This array corresponds to the registered fonts array which stores the font-header information. Before you can use the **LoadFont%** function, you must use the **RegisterFont%** function.

Note The total size of the font data should be less than or equal to 64K unless the /AH option is used when invoking QBX. The /AH option allows the use of larger arrays.

For more information on selecting a particular loaded font, see the **SelectFont** procedure.

Example The following example demonstrates the **LoadFont%** function.

```
r% = RegisterFonts("tmsrb.fon")      ' r% should = 6 after call.
l% = LoadFont("h8/h12/h24")          ' l% should = 3 after call.
' 3 fonts (8,12,24 pts) will be loaded and usable.
SelectFont 2                         ' The 12-point font is current.
SetGTextColor 10                     ' Color 10 is light green.
textlen% = OutGText (1,1,"TmsRmn 12-point")
END
```

Note that to load fonts according to their order in the font file, `l%` would take the form:

```
l% = LoadFont%("n1,n3,n6")    ' Load the first, third, and sixth fonts.
```

OutGText% FUNCTION

Action Outputs text in the currently selected font, using the current graphics color at the current graphics cursor position, and returns the pixel length of the character output.

Syntax **OutGText%** (*x*, *y*, *txt$*)

Remarks The **OutGText%** function uses the following arguments:

Argument	Description
x	A single-precision number that is the x coordinate of the upper-left boundary of the first character.
y	A single-precision number that is the y coordinate of the upper-left boundary of the first character.
txt$	Text string to output.

After the **OutGText%** function has completed outputting the text, it returns the pixel length of the character string, and *x* and *y* are changed to the upper-right position of the last character.

RegisterFonts% FUNCTION

Action Registers the font-header information from a specified .FON file and returns the number of fonts registered.

Syntax **RegisterFonts%** (*filename$*)

Remarks The **RegisterFonts%** procedure registers the font-header information from the .FON file specified in *filename$*. The font-header information is stored in the registered-fonts array. This array corresponds to the loaded-fonts array which contains the character-bit maps of the fonts stored in the .FON file. Because an .FON file may hold one or more fonts, **RegisterFonts%** returns the number of fonts registered by the function.

This procedure will register any number of fonts up to the maximum set with the **SetMaxFonts** procedure or to the limit of available memory.

Note When using the **RegisterFonts%** procedure, be sure you have access to the .FON file on disk so that it can find the filename of the currently selected font.

For more information on loading the fonts, see the **LoadFont%** procedure.

RegisterMemFont% FUNCTION

Action Registers the font-header information of fonts that reside in memory and returns the number of fonts registered.

Syntax **RegisterMemFont%** (*fontseg%*, *fontoffset%*)

Remarks The **RegisterMemFont%** procedure uses the following arguments:

Argument	Description
fontseg%	An integer that is the segment address of the in-memory font.
fontoffset%	An integer that is the offset address of the in-memory font.

The **RegisterMemFont%** function returns either 0 or 1 because only one in-memory font may be registered each time **RegisterMemFont%** is called. Any font that you use or create must define the font according to the font-file specifications for Microsoft Windows applications.

To get the segment and offset addresses, call the **DefaultFont** assembly-language routine in the Presentation Graphics toolbox. This assembly-language routine initializes the data that defines the Windows-type font file and returns the segment and offset addresses of the font file structure. The Windows-type font file must be loaded into memory or initialized before you can call the **RegisterMemFont%** procedure.

The **RegisterMemFont%** procedure enables a user to build a font into the executable module so the font does not have to be read in by the program at execution time, which requires the font file to be on disk, in addition to the executable file.

Note BASIC relocatable data structures cannot be used to hold font information because they do not have a fixed memory location.

SelectFont SUB

Action Designates a loaded font as the active font.

Syntax **SelectFont** (*n*)

Remarks The argument *n* contains the number of the loaded font that is selected and becomes the active font.

If *n* is greater than the maximum number of loaded fonts, the procedure automatically selects the font number that corresponds to the remainder from the modular division between *n* and the number of currently loaded fonts. For example, if *n* is 12 and the maximum of loaded fonts is 8, then the font number selected is 4.

SetGCharSet SUB

Action Sets the character set used in subsequent graphics characters.

Syntax **SetGCharSet** *(charset%)*

Remarks The argument *charset%* is the character set used in mapping input characters to output characters. *charset%* is an integer with two possible values: cIBMChars and cWindowsChars. The two values are constants defined in the FONTB.BI file.

SetGTextColor SUB

Action Sets the character color used in subsequent graphic characters.

Syntax **SetGTextColor** (*color%*)

Remarks The argument *color%* is an integer that is the pixel value of the color to use in graphic characters.

SetGTextDir SUB

Action Sets the horizontal or vertical orientation of graphics characters.

Syntax **SetGTextDir** (*dir%*)

Remarks The argument *dir%* is an integer that specifies a direction for the graphics characters. Possible values for *dir%*, and the resulting character orientation (counterclockwise rotation) are:

Value	Character direction
0	0 degrees (no rotation)
1	90 degrees (top of character is to the left)
2	180 degrees (characters are inverted)
3	270 degrees (top of character is to the right)

SetMaxFonts SUB

Action Sets the maximum number of fonts that are allowed to be registered and loaded.

Syntax **SetMaxFonts** (*maxregistered%*, *maxloaded%*)

Remarks The **SetMaxFonts** procedure uses the following arguments:

Argument	Description
maxregistered%	An integer that specifies the maximum number of fonts that can be registered.
maxloaded%	An integer that specifies the maximum number of fonts that can be loaded.

The **SetMaxFonts** procedure allocates a maximum number of font-header elements to hold font information. The font information is registered when the **RegisterFonts%** function is called. Therefore, **SetMaxFont** is called prior to **RegisterFonts%** or **LoadFont%**. If **SetMaxFonts** is not called, the default number is 10.

SetMaxFonts acts as a complete reinitialization by removing any previously registered or loaded fonts from memory and then setting the new maximum limits of fonts that can be loaded or registered.

UnRegisterFonts SUB

Action Removes registered fonts from memory.

Syntax **UnRegisterFonts** ( )

Remarks The **UnRegisterFonts** procedure reduces the array storing the font headers to one element, thereby removing the registered fonts from memory.

Note When you use **UnRegisterFonts** to remove the font-header information from the registered fonts array, you reduce the array to only one position. Therefore, if you want to re-register fonts, you must call **SetMaxFonts** to reset the maximum number of fonts that can be registered; for example, `SetMaxFonts(6,6)`, before calling the **RegisterFonts%** function.

The User Interface Toolbox

The User Interface toolbox is a group of BASIC procedures and assembly-language routines that is included with Microsoft BASIC. Together, these tools allow you to add professional-looking windows and menus to your programs.

The User Interface toolbox includes many of the user interface features of the QBX programming environment. When you use the QBX user interface, you are using many of the features that the toolbox can bring to your own programs.

- You can define pull-down menus to support:
  - Color
  - Mouse
  - Access keys
  - Shortcut keys
- You can choose these options in overlapping character-based windows:
  - Color
  - Closable windows
  - Movable windows
  - Resizable windows
  - Title bars
  - Window-border character
- You can use list boxes and alert boxes.
- You can specify dialog boxes with:
  - Edit fields
  - Option buttons
  - Command buttons
  - Area buttons

The BASIC source code to all of the toolbox procedures is provided with Microsoft BASIC. You can use these procedures as listed, or alter or expand their functionality to meet your requirements. Even though most of the procedures are written in BASIC, they are fast enough for most applications. There is also an assembly-language object routine, UIASM.OBJ, to ensure that you get quick response from your user-interface design. This assembly-language module is linked into the libraries and Quick libraries created from the User Interface toolbox.

What Makes Up the User Interface Toolbox

The User Interface toolbox actually consists of four parts — that is, four separate BASIC source-code files — and an include file for each. The files are:

- MENU.BAS, MENU.BI
- WINDOW.BAS, WINDOW.BI
- MOUSE.BAS, MOUSE.BI
- GENERAL.BAS, GENERAL.BI

In each case, the .BAS file contains the actual code for the procedures and the .BI file contains user-defined type definitions and procedure declarations used in the .BAS file.

The following discussion briefly explains the contents of each part of the User Interface toolbox.

Menus

The MENU.BAS source-code file contains all the procedures that let you create custom menus in your programs. These procedures are dependent on the other parts of the User Interface toolbox (GENERAL.BAS and MOUSE.BAS). If you use the procedures in MENU.BAS, you must include GENERAL.BI, MOUSE.BI, and MENU.BI in your program so you have the proper declarations and definitions. In addition to procedure declarations, the header file MENU.BI contains definitions of several user-defined types that are used in global variables to store menu configuration information.

A more detailed explanation of menus and their application is described in the MENU.BAS section below.

Windows

The WINDOW.BAS source-code file contains the procedures that let you create custom character-based windows in your programs. WINDOW.BAS is dependent on all the other parts of the User Interface toolbox; therefore, when you use the procedures in WINDOW.BAS, you must include GENERAL.BI, MOUSE.BI, MENU.BI, and WINDOW.BI so that all declarations and definitions on which WINDOW.BAS depends are available.

The header file WINDOW.BI contains procedure declarations along with the definitions of user-defined types. These user-defined types are used to store information about the characteristics of windows, buttons, and edit fields.

Mouse

The MOUSE.BAS source-code file provides mouse support. The procedures in this code can be used by themselves if need be, but they were designed as an integral part of the User Interface toolbox. The routines specifically support the Microsoft Mouse. A Microsoft Mouse and driver are recommended. However, the procedures support any pointing device with a 100 percent Microsoft-Mouse-compatible driver. A Microsoft driver (MOUSE.COM) is shipped with BASIC.

If you use the procedures in MOUSE.BAS, you must include GENERAL.BI and MOUSE.BI in your program so you will have the proper declarations and definitions. The header file MOUSE.BI includes procedure declarations for the MOUSE.BAS file.

General

The GENERAL.BAS source-code file contains a number of general purpose character-based routines that are used in both the MENU.BAS and WINDOW.BAS files. While these files can be used by themselves, they are intended to support the other procedures in the User Interface toolbox. If you use any of the procedures in GENERAL.BAS, you must include the header file, GENERAL.BI.

The header file GENERAL.BI contains a user-defined type definition (RegType) that is used with the **Interrupt** routine to access DOS interrupt services for mouse support. This DOS interrupt also returns the shift state of special keys, and scrolls screen areas. GENERAL.BI contains a number of global constant definitions that apply throughout the User Interface toolbox. These definitions are described in the following table:

Constant	Value	Comment
FALSE	0	
TRUE	–1	
MINROW	2	Minimum number of rows on screen
MAXROW	25	Maximum number of rows on screen
MINCOL	1	Minimum number of columns on screen
MAXCOL	80	Maximum number of columns on screen
MAXMENU	10	Maximum number of menus on menu bar
MAXITEM	20	Maximum number of items on a menu
MAXWINDOW	10	Maximum number of windows on screen
MAXBUTTON	50	Maximum number of buttons on screen
MAXEDITFIELD	20	Maximum number of edit fields on screen
MAXHOTSPOT	20	Maximum number of active areas on screen

Note An active area is defined as any area that can be selected by the user, including all types of buttons and edit fields, and any of the special characters used to affect windows.

How to Use the Toolbox in Your Programs

You can incorporate the User Interface toolbox into your programs in three ways. The first is to simply include the source-code files you plan to use in your program. For each source-code file you include, be sure to also include the associated .BI file. Use the **$INCLUDE** metacommand to include external files in your program.

The second way is to load each BASIC source-code file as a separate module and use a Quick library only for the assembly-language routines. You should use this method if you are planning to alter or expand the functionality of the User Interface toolbox. Create the Quick library you need with the following command:

```
LINK /Q UIASM.OBJ QBX.LIB, UIASM.QLB,,QBXQLB.LIB;
```

The final method—and most practical when creating stand-alone programs—is to create a Quick library that uses the procedures in the BASIC source-code files and the assembly-language files. This is somewhat more difficult, but is still quite easy. If you chose to have Quick libraries created when you first installed BASIC, you already have the Quick library you need. If you didn't, you'll have to make it yourself. To begin, be sure that you are in the QBX directory where all of the .LIB files that came with the program are located.

Creating a complete Quick library involves several steps. First you'll have to create a Quick library that includes needed procedures that are already in the QBX libraries (QBX.QLB and QBX.LIB) plus the User Interface toolbox object file, UIASM.OBJ. You can do this at the system prompt with the following command:

```
LINK /Q UIASM.OBJ QBX.LIB, UIASM.QLB,,QBXQLB.LIB;
```

Next, create the parallel .LIB library using the same files you just used to make the Quick library. Use the following command:

```
LIB UIASM.LIB+UIASM.OBJ+QBX.LIB;
```

Once you've created the first two libraries, you can create the libraries that include not only the assembly-language routines, but the BASIC procedures as well. The easiest way is to do this from the QBX programming environment. Start QBX with the /L option and specify the Quick library that you just created, as follows:

```
QBX /L UIASM.QLB
```

Once you are in the QBX programming environment, use the File menu to load each of the individual files you want to include in your User Interface toolbox library. For example, if you want to use menus in your program, load GENERAL.BAS, MOUSE.BAS, and MENU.BAS. GENERAL.BAS and MOUSE.BAS are required because of dependencies that exist between the components of the User Interface toolbox. If you want windows, load GENERAL.BAS, MOUSE.BAS, MENU.BAS, and WINDOW.BAS.

With all of the files loaded, choose Make Library from the Run menu. When prompted for a library name, use the name UITBEFR.QLB. Press Return and two libraries will be created for you. The first is the Quick library that you will use whenever you want to incorporate menus into your programs that run in QBX. The second is the parallel .LIB file that will allow you to create a stand-alone .EXE file from your BASIC program.

If you change any of the BASIC code that comprises the User Interface toolbox, you can relink those object modules and create new libraries and Quick libraries as needed. More complete information about creating libraries and using mixed-language programming can be found in Chapter 18, "Using LINK and LIB" in the *Programmer's Guide*.

If you use a library or Quick library that contains procedures from one or more of the User Interface toolbox source-code files, you must include the .BI file for that source-code file into your program before you call any of the library's procedures. If you include a complete source-code file, it is not necessary to include the .BI file, because an **$INCLUDE** metacommand is already in the source-code file. However, if you include the source-code file in your program, you must ensure that you also include any other files (either .BAS or .BI) that the included code might depend on. For example, if you use WINDOW.BAS, you must also include all of the other parts of the User Interface toolbox. The procedures in WINDOW.BAS depend on definitions, declarations, and procedures in each of the other parts.

Detailed Description

The following sections contain a detailed descriptions of each of the individual parts of the User Interface toolbox. Because the User Interface toolbox is character based, many procedures require row and column coordinates as arguments. Where these are required, the following definitions apply:

Argument	Description
row%, *col%*	An integer pair that describes a screen position.
row1%, *col1%*	An integer pair that describes the upper-left corner of an area.
row2%, *col2%*	An integer pair that describes the lower-right corner of an area.

Note

The *row%* and *col%* coordinates can be actual screen coordinates. However, often they are coordinates that are relative to the upper-left corner of the current window.

MENU.BAS

Using the procedures in MENU.BAS is straightforward. In most cases, the demonstration program UIDEMO.BAS provides enough information for a BASIC programmer of some experience to get started. However, a few notes of introduction are in order.

When you use the procedures in MENU.BAS, you must provide the following global array declarations in your program, in the order shown:

```
COMMON SHARED /uitools/GloMenu AS MenuMiscType
COMMON SHARED /uitools/GloTitle() AS MenuTitleType
COMMON SHARED /uitools/GloItem() AS MenuItemType
DIM GloTitle(MAXMENU) AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM) AS MenuItemType
```

The order is important because the block defined by **COMMON** in the Quick library must agree with the block defined by **COMMON** in the programming environment. These global arrays are used to store information about the menus you define.

As previously mentioned, the menus in the User Interface toolbox work like those in the QBX programming environment. Menu titles are displayed on a menu bar that extends across the top of the screen in row 1. Associated with each menu are one or more menu items. From the programmer's standpoint, menus are numbered from left to right, beginning at 1. Menu items—that is, those choices associated with each menu—are numbered from top to bottom, also beginning at 1. For any given menu, menu item 0 is always the menu title.

The **MenuColor** procedure lets you select the color scheme for your menus. A single **MenuSet** statement is all that is required to set up each menu title on the menu bar or each menu item associated with a menu.

Menus can be selected with the mouse—by pointing and clicking—or with the keyboard, by pressing the Alt key and the highlighted letter in the desired menu title. When a menu is selected, the choices associated with that menu will pull down, or drop, from the menu bar so that you can see the choices available. When a pull-down menu is displayed, menu items can be chosen by pointing and clicking with the mouse, or pressing the highlighted key associated with that choice. The key combinations used in the latter selection technique are called access keys. Access keys consist of a series of menu-item-selection keystrokes; each series begins by pressing the Alt key.

You also can set up shortcut keys to use in lieu of the mouse or the regular access keys. Shortcut keys can use any character or extended character, except those that use the Alt key. For example, you can use Ctrl+X, Ctrl+F10, or simply F3). To enable a shortcut key, use a **ShortCutKeySet** statement to identify the menu and menu item with the definition of the shortcut key.

Before actually building your menus, you need to initialize global arrays to speed up menu response. Use the **MenuInit** procedure to initialize the arrays you declared and initialize the mouse.

The following code fragment illustrates how to put together a short menu with one title, two item choices and a means of exiting the menu:

```
MenuInit        ' Initialize everything, including the mouse.
MenuSet 1,0,1,"Menu Title",1          ' Set up menu title.
MenuSet 1,1,1,"Selection 1 F1",11     ' Set up first selection.
MenuSet 1,2,1,"Selection 2 F2",11     ' Last parameter determines
                                      ' highlight.
MenuSet 1,3,1,"-",1                   ' Make a line across the menu.
MenuSet 1,4,1,"Exit Ctrl-X",2         ' Set up way to get out.

ShortCutKeySet 1,1,CHR$(0)+CHR$(59)   ' Set F1 as shortcut key for 1.
ShortCutKeySet 1,2,CHR$(0)+CHR$(60)
ShortCutKeySet 1,4,CHR$(24)           ' Set Ctrl-X for Exit.

MenuColor 14,1,12,8,14,3,12           ' Specify color scheme
MenuPreProcess                        ' Do all necessary calculations.
MenuShow                              ' Display the menu.
MouseShow                             ' Make mouse cursor visible.
```

Notice that the menu title is described as menu 1, item 0. Item 0 means that the **MenuSet** statement specifies a menu title, not a menu item. Numbers from 1 through 4 are the remaining selections. Item 3 is special. It's not really a selection item. The hyphen character (-) causes a bar to be placed across the pull-down menu. It is useful for breaking up functional groupings of menu selections. You can't select it, so you should remember to use a **SELECT CASE** statement that excludes that item number when processing the selections. See the demonstration program UIDEMO.BAS for a working example.

Also notice that within the text for Selection 1, F1 is shown to the right. F1 is the shortcut key for Selection 1 that is programmed using the first **ShortCutKeySet** statement. The extended keyboard code for the F1 key is CHR$(0) + CHR$(59).

Use **MenuColor** to set up the colors for your menus, the highlighted colors to use when an item is selected, and the highlighted characters to use when choosing a menu item from the keyboard.

MenuPreProcess performs the necessary calculations and builds indexes that help speed up menu execution. **MenuShow** actually displays the menu bar on the screen. The three-dimensional shadow and other programming chores are handled for you automatically. Once the menu bar is displayed, the call to **MouseShow** makes the mouse cursor visible.

After you've displayed your menus, you want to be able to have the program translate the available choices into action. Processing menu selections is also very straightforward. The **MenuInkey$** procedure within a **WHILE...WEND** control-flow structure serves well to poll user input. The following code fragment illustrates how to process user input and translate a user's choice to a desired action:

```
Finished = FALSE
WHILE NOT Finished
   kbd$ = MenuInkey$          ' Monitor all user input.
   SELECT CASE kbd$           ' See what the user entered.
   CASE "menu"                ' If kbd$ = "menu".
      menu = MenuCheck(0)     ' Get menu number.
      item = MenuCheck(1)     ' Get item number.
      GOSUB HandleMenuEvent   ' Handle selection.
   CASE ELSE
      .
      .                       ' If kbd$ = anything else.
      .
   END SELECT
WEND

HandleMenuEvent:
SELECT CASE menu
CASE 1                        ' Handles Menu 1 .
   SELECT CASE item
   CASE 1
      Selection1Routine       ' Handle selection 1.
   CASE 2
      Selection2Routine       ' Handle selection 2.
   CASE 4
      Finished = TRUE         ' Handle Exit.
   END SELECT
END SELECT
RETURN
```

The code shown above processes keyboard input and mouse input with **MenuInkey$**. If you wanted to preclude any keyboard input, you could use **MenuEvent** instead of **MenuInkey$**, as follows:

```
WHILE NOT Finished
   MenuEvent                  ' Monitor menu events only.
   IF MenuCheck(2) THEN
      menu = MenuCheck(0)     ' Get menu number.
      item = MenuCheck(1)     ' Get item number.
      GOSUB HandleMenuEvent   ' Determine which item.
   END IF
   .                          ' What to do if MenuCheck(2)
   .                          ' hasn't changed.
   .
WEND
```

The preceding information is intended to give you an overview of how the menus work. With this information and the detailed information provided for each procedure, you should be able to incorporate menus into your BASIC applications. Be sure to see the example code in UIDEMO.BAS that demonstrates how most of the procedures are used.

A description of each procedure that comprises MENU.BAS is next.

MenuCheck FUNCTION

Action Returns an integer that indicates which menu selection, if any, was made following a menu or shortcut-key event.

Syntax *variablename%* = **MenuCheck**(*action%*)

Remarks The **MenuCheck** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
action%	An integer that identifies the specific information requested. Values are as follows:

Value	Associated action/Return value
0	Checks to see if a menu item was selected since **MenuCheck** was last called. If a menu item was selected, **MenuCheck** returns the number of the menu and sets the global variables so that when `MenuCheck(1)` is called, the appropriate menu item number is returned. If no menu item was selected, returns 0.
1	Returns the matching menu item of the last menu selection. `MenuCheck(1)` is reset each after each call to `MenuCheck(0)`.
2	Returns TRUE (–1) if menu item has been selected as of the last `MenuCheck(0)` call, or FALSE (0) if not. If a selection has occurred, use `MenuCheck(0)` and `MenuCheck(1)` to determine what selection was made. `MenuCheck(2)` changes no values.

The **MenuCheck** procedure simulates polling for a menu event. When a menu event occurs, global variables are set to their appropriate values. `MenuCheck(0)` then returns the menu number, or 0 if no menu was selected since the last call to **MenuCheck**.

You can use **MenuCheck** to provide information about menu selection as follows:

```
kbd$ = MenuInkey$
SELECT CASE kbd$
CASE "menu"
   menu = MenuCheck(0)
   item = MenuCheck(1)
   MouseHide
   PRINT menu, item
   MouseShow
CASE ELSE
   PRINT kbd$
END SELECT
```

or

```
MenuEvent
IF MenuCheck(2) THEN
   menu = MenuCheck(0)
   item = MenuCheck(1)
   MouseHide
   PRINT menu, item
   MouseShow
END IF
```

See Also **MenuEvent, MenuInkey$, ShortCutKeyEvent**

Example See the main loop of the code example UIDEMO.BAS for a practical implementation of the **MenuCheck** procedure.

MenuColor SUB

Action Assigns colors to the components of a menu.

Syntax **MenuColor** *fore%*, *back%*, *highlight%*, *disabled%*, *cursorfore%*, *cursorback%*, *cursorhi%*

Remarks The **MenuColor** procedure uses the following arguments:

Argument	Description
fore%	An integer that defines menu foreground color (0–15).
back%	An integer that defines menu background color (0–7).
highlight%	An integer that defines text color (0–15) of the access key character.
disabled%	An integer that defines text color of disabled items (0–15).
cursorfore%	An integer that defines the menu cursor foreground color (0–15).
cursorback%	An integer that defines the menu cursor background color (0–7).
cursorhi%	An integer that defines text color (0–15) of the access key character when the menu cursor is on that menu item.

The values specified are placed into global variables. You can use any color scheme you choose within the range of available colors. Shadows that appear beneath menu windows are always black.

Use **MenuColor** in your program to set up menu colors. Colors should be initialized before the menu is displayed the first time. If colors are not initialized, the default monochrome color scheme places the following integers into the argument list (in order): 0, 7, 15, 8, 7, 0, 15. This results in a monochrome display.

If you change colors while your program is running, use **MenuShow** to show the changes.

See Also **MenuShow**

Example See the SetupMenu procedure in the code example UIDEMO.BAS for the specific implementation of the **MenuColor** procedure.

MenuDo SUB

Action Used internally by other procedures in MENU.BAS. Do not use or alter this procedure.

Syntax **MenuDo**

Remarks The **MenuDo** procedure is used only within MENU.BAS to process menu selection using the mouse and keyboard. To process menu selections in your program, you should use **MenuInkey$** or **MenuEvent**, both of which use this procedure.

If menus are enabled (using **MenuOn**), **MenuDo** takes control of the program when the user is selecting a menu item by any means. **MenuDo** controls the display and responds to user input until a selection has been made. Once a selection has been made, use the **MenuCheck** procedure to determine which menu item was selected.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **MenuEvent**, **MenuInkey$**

MenuEvent SUB

Action Monitors user input to determine if a menu item is being chosen, with either the mouse or the keyboard.

Syntax **MenuEvent**

Remarks If **MenuEvent** detects either a pressed Alt key or a mouse button press while the mouse cursor is on the top row, program control is transferred to **MenuDo**. **MenuDo** retains program control until an item is selected or the user aborts the menu process either by releasing the mouse button somewhere other than the top row or by pressing the Esc key. Once an item is selected, use **MenuCheck** to evaluate the selection.

MenuEvent is used by the **MenuInkey$** procedure to determine when a user has pressed the Alt key in conjunction with a keyboard selection or has selected a menu by moving the mouse cursor to the top row and pressing the left mouse button.

See Also **MenuInkey$**, **ShortCutKeyEvent**

MenuInit SUB

Action Initializes global menu arrays and the mouse driver-servicing routines.

Syntax **MenuInit**

Remarks To enable mouse driver servicing, **MenuInit** calls **MouseInit** in MOUSE.BAS.

Use **MenuInit** at or near the beginning of any program that uses the menu procedures provided in the User Interface toolbox. You must call **MenuInit** before any other menu procedures are used.

See Also **MenuPreProcess**

Example See the initialization section in the code example UIDEMO.BAS for a specific implementation.

MenuInkey$ FUNCTION

Action Performs a BASIC **INKEY$** procedure, as well as both a **MenuEvent** procedure and a **ShortCutKeyEvent** procedure, returning a string.

Syntax *variablename$* = **MenuInkey$**

Remarks The **MenuInkey$** procedure operates exactly like the standard BASIC **INKEY$** procedure, but additionally performs **MenuEvent** and **ShortCutKeyEvent** procedures. If either a menu event or a shortcut-key event occurs, **MenuInkey$** returns the word "menu" instead of the normally expected **INKEY$** value. When **MenuInkey$** returns the word "menu," use **MenuCheck** to determine which menu and menu item was selected.

See Also **MenuDo, MenuEvent, ShortCutKeyEvent**

Example See the main loop in the code example UIDEMO.BAS for a specific implementation.

MenuItemToggle SUB

Action Toggles the state of of the selected menu item between 1 (enabled) and 2 (enabled with a check mark) to indicate selection of the menu item.

Syntax **MenuItemToggle** *menu%*, *item%*

Remarks The **MenuItemToggle** procedure uses the following arguments:

Argument	Description
menu%	An integer that identifies the position (from left to right) of the affected menu.

item% — An integer that identifies the number (from top to bottom) of the menu item within the menu.

Use **MenuItemToggle** whenever you want to indicate that a particular menu item has been selected. It is not necessary to re-execute **MenuPreProcess** after performing a **MenuItemToggle** operation.

See Also **MenuPreProcess, MenuSetState**

MenuOff SUB

Action Turns off menu and shortcut-key event processing.

Syntax **MenuOff**

Remarks Menu and shortcut-key event processing is disabled by setting the global variable GloMenu.MenuOn to FALSE (0). When menu event processing is off, **MenuDo** ignores any mouse or keyboard activity that would normally trigger a menu event.

Use **MenuOff** anytime you want to turn menu event processing off.

See Also **MenuOn**

MenuOn SUB

Action Turns on menu and shortcut-key event processing.

Syntax **MenuOn**

Remarks Menu and shortcut-key event processing is enabled by setting the global variable GloMenu.MenuOn to TRUE (–1). When menu event processing is on, **MenuDo** takes control of the display and the program when any mouse or keyboard activity occurs to trigger a menu event.

Use **MenuOn** anytime you want to turn menu event processing back on after disabling with **MenuOff**.

See Also **MenuOff**

MenuPreProcess SUB

Action Performs calculations and builds indexes so the menu procedures run faster.

Syntax **MenuPreProcess**

Remarks Use **MenuPreProcess** anytime **MenuSet** is performed one or more times, and any time the state of a menu item is changed using **MenuSetState** to change to or from the empty (–1) or not displayed state. You must call **MenuPreProcess** anytime changes other than item selection changes are made to the menus.

See Also **MenuSet, MenuSetState**

Example See the **SetupMenu** procedure in the code example UIDEMO.BAS for a specific implementation.

MenuSet SUB

Action Defines the structure of your menus and defines the access keys associated with individual menu selections.

Syntax **MenuSet** *menu%, item%, state%, text$, accesskey%*

Remarks The **MenuSet** procedure uses the following arguments:

Argument	Description
menu%	An integer that identifies the position (from left to right) of a menu on the menu bar.
item%	An integer that identifies the position (from top to bottom) of the menu item within the menu. If *item%* is 0, then *text$* is the menu title. Other numbers indicate consecutive menu selections.
state%	An integer that indicates the state of the menu item:

Value	Menu state that appears on menu
0	Disabled—appears in color defined by the *disabled%* variable set with **MenuColor.**
1	Enabled—this is the normal state.
2	Enabled with a check mark.

text$	A string that is the name of the menu item. Menu titles are limited to 15 characters; individual menu items are limited to 30 characters.
accesskey%	An integer that indicates the position, within *text$*, of the character that is used to choose the menu item. This access key is highlighted in color defined by the *highlight%* variable set with **MenuColor**.

Use **MenuSet** to initialize the contents of your menus. A separate **MenuSet** entry is required for each item, including the title on each menu. Each **MenuSet** entry defines either a menu title or a menu item that is to be displayed and the state of individual menu items. Menu items can be enabled, disabled, checked, or even made invisible. You also can specify access keys for menu items to allow one-stroke selection.

MenuPreProcess must be executed after a series of **MenuSet** statements or any time the state of a menu item is changed using **MenuSetState**. **MenuShow** must be re-executed if any change to the menu title has occurred.

See Also **MenuPreProcess, MenuSetState, MenuShow**

Example See the **SetupMenu** procedure in the code example UIDEMO.BAS for a specific implementation.

MenuSetState SUB

Action Explicitly assigns the state of a menu item.

Syntax **MenuSetState** *menu%*, *item%*, *state%*

Remarks The **MenuSetState** procedure uses the following arguments:

Argument	Description
menu%	An integer that identifies the position (from left to right) of the affected menu.
item%	An integer that identifies the number of the menu item within the menu.

state% An integer that indicates the state of the menu item:

Value	Menu state
–1	Empty—does not appear on menu.
0	Disabled—appears in color defined by the *disabled%* variable set with **MenuColor.**
1	Enabled—this is the normal state.
2	Enabled with a check mark.

Use **MenuSetState** whenever you want to change the state of a particular menu item. You must re-execute **MenuPreProcess** if the state of the menu item changes to or from the empty or not displayed state.

See Also **MenuPreProcess**

MenuShow SUB

Action Draws the menu bar across the top of the screen.

Syntax **MenuShow**

Remarks All menu titles that have been established with **MenuSet** are displayed. Use **MenuShow** to initially display your menu bar once it has been defined with **MenuSet**, and use it each time you change the menu bar.

See Also **MenuPreProcess**

Example See the initialization section in the code example UIDEMO.BAS for a specific implementation.

ShortCutKeyDelete SUB

Action Deletes the shortcut key associated with a particular menu item.

Syntax **ShortCutKeyDelete** *menu%*, *item%*

Remarks The **ShortCutKeyDelete** procedure uses the following arguments:

Argument	Description
menu%	An integer that identifies the position (from left to right) of the affected menu.
item%	An integer that identifies the number of the menu item within the menu.

Use **ShortCutKeyDelete** anytime you want to void shortcut keys that were previously established with **ShortCutKeySet**.

See Also **ShortCutKeyEvent, ShortCutKeySet**

ShortCutKeyEvent SUB

Action Polls user input to determine if a menu item is being selected by using one of the shortcut keys.

Syntax **ShortCutKeyEvent** *kbd$*

Remarks The argument *kbd$* is a string that contains a character entered at the keyboard. **ShortCutKeyEvent** checks *kbd$* against a list of shortcut keys defined using **ShortCutKeySet**. If a match is found, the proper menu and item are selected. This duplicates exactly the functionality of the **MenuEvent** procedure. Use **MenuCheck** to determine which menu and item were selected.

If you use **MenuInkey$** in your program, **ShortCutKeyEvent** is automatically performed; if you use the BASIC **INKEY$** procedure, you must explicitly call **ShortCutKeyEvent** to process shortcut-key events.

See Also **MenuCheck, ShortCutKeyDelete, ShortCutKeySet**

ShortCutKeySet SUB

Action Assigns shortcut keys to individual menu items.

Syntax **ShortCutKeySet** *menu%, item%, kbd$*

Remarks

The **ShortCutKeySet** procedure uses the following arguments:

Argument	Description
menu%	An integer that identifies the position (from left to right) of the affected menu.
item%	An integer that identifies the number of the menu item within the menu.
kbd$	A string that indicates which character is allowed as a shortcut key for selection of the menu item. The string argument *kbd$* can be any single character, or any extended characters, except those that use the Alt key.

Without shortcut keys, each menu item can be selected by pressing the Alt key and the highlighted character in the menu name, followed by the highlighted letter in the menu item. **CommandKeySet** provides additional functionality by allowing you to specify a single character, such as Shift+F1 or F1, as a shortcut to menu item selection. A separate **ShortCutKeySet** entry is required for each shortcut key you assign.

Use **ShortCutKeySet** statements during menu initialization at the same place where **MenuSet** is used.

See Also

MenuSet, **ShortCutKeyDelete**, **ShortCutKeyEvent**

Example

See the **SetupMenu** procedure in the code example UIDEMO.BAS for a specific implementation.

WINDOW.BAS

As with MENU.BAS, you can learn much about using the procedures in WINDOW.BAS by studying the demonstration program UIDEMO.BAS, which is provided as a code example. However, a few words of introduction will be helpful. The best way to get a feel for the ease of use of the window procedures is to experiment for yourself, using the code in the demonstration as an example.

In addition to the global-array declarations used with MENU.BAS, you must declare the following global arrays in your program, in the order shown, if you use the procedures in WINDOW.BAS:

```
COMMON SHARED /uitools/GloWindow()   AS WindowType
COMMON SHARED /uitools/GloButton()   AS ButtonType
COMMON SHARED /uitools/GloEdit()     AS EditFieldType
COMMON SHARED /uitools/GloStorage    AS WindowStorageType
COMMON SHARED /uitools/GloBuffer$()
DIM GloWindow(MAXWINDOW)             AS WindowType
DIM GloButton(MAXBUTTON)             AS ButtonType
DIM GloEdit(MAXEDITFIELD)            AS EditFieldType
DIM GloWindowStack(MAXWINDOW) AS INTEGER
DIM GloBuffer$(MAXWINDOW +1,2)
```

These global arrays are used to store information about the windows you plan to use in your program. The window effects provided by WINDOW.BAS let you open and use text windows virtually anywhere on your screen. For each window, you can independently define the following window characteristics:

- Initial position, by row and column
- Color
- Ability to close window or not
- Ability to move window or not
- Ability to resize window or not
- Window title
- Border characters

You also can define as modal windows any of the windows you make. A modal window is a window that forces the user to respond to a prompt within the window itself. You cannot select, either with the mouse or the keyboard, anything outside the window's border. An alert message box is an example of a modal window.

A single **WindowOpen** statement lets you open your window. Once you've opened a window, you can fill it with text, edit fields, list boxes with scroll bars, and push buttons of several types, all of which behave much like the QBX user interface. If you open several windows, you can move between them by using the mouse to select the one you want. A single procedure, **WindowDo**, monitors your mouse and keyboard activity while a window is open, and responds according to your input.

To put text in whatever window is active, **WindowLocate** lets you specify where the text is to go and **WindowPrint** lets you print it there. Edit fields (boxes that contain some text for you to either accept or change) are created in a window using **EditFieldOpen**. They are closed with **EditFieldClose**.

List boxes with scroll bars are displayed by passing an array containing the items in the list to the **ListBox** procedure. In addition to a list of items, List boxes always contain OK and Cancel command buttons.

An assortment of buttons, with a full range of features, are placed on the screen with the **ButtonOpen** procedure. Supported button types and their behavior are described in the following table:

Button type	**Explanation**
Command button	Confirms settings, gets help, or escapes.
Option button	Selects one of several options (use direction keys).
Check box	Turns one option on or off (use Spacebar).
Area button	An invisible button on the screen that occupies a defined area. When selected, causes some event to occur.
Vertical scroll bar	Vertically scrolls long lists or screen areas.
Horizontal scroll bar	Horizontally scrolls screen areas that are wider than the active window.

Before opening a window—in fact, before using any of the routines in WINDOW.BAS—you must do some program initialization. The following code fragment illustrates how to do the initialization and how to use some of the procedures in WINDOW.BAS:

```
MenuInit                                  ' Initialize menus and mouse.
WindowInit                                ' Initialize windows.
MouseShow                                 ' Make mouse cursor visible.

WindowOpen 1,5,25,15,45,0,3,0,3,0,-1,-1,0,0,0,"-Window 1-"
WindowPrint 1,""                          ' Put a blank line at the top.
WindowPrint 1,"Features:"                 ' Put text in the window.
WindowPrint 1,"Title bar"
WindowPrint 1,"Closable window"
WindowPrint 1,"No border"
```

```
ButtonOpen 1,2,"OK",11,8,1,0,1       ' Open an OK button.

WindowOpen 2,8,29,17,49,0,5,0,5,15,-1,-1,0,0,1,"-Window 2-"
WindowPrint 1, ""                    ' Put a blank line at the top.
WindowPrint 1, "Features:"           ' Put text in the window.
WindowPrint 1, "Title bar"
WindowPrint 1, "Closable window"
WindowPrint 1, "Single-line border"
WindowLine 9                         ' Put a line across the bottom.
ButtonOpen 1,2,"OK",11,8,1,0,1

ExitFlag = FALSE

WHILE NOT ExitFlag
   WindowDo 1,0                      ' Monitor for a window event.
   SELECT CASE Dialog(0)             ' Return what event occurred.
   CASE 1,4,6                        ' Handle user actions:
      WindowClose WindowCurrent      ' Selecting OK, selecting the
                                     ' close box, or pressing Enter.
      IF WindowCurrent = 0 THEN      ' Just in case it's the last
         ExitFlag = TRUE             ' or only window.
      END IF
   CASE 3                            ' Handle selecting another window
      WindowSetCurrent Dialog(3)     ' and making it current.
   CASE 9                            ' Handle pressing Esc.
      ExitFlag = TRUE
   CASE ELSE                         ' Handle everything else.
   END SELECT
WEND

WindowClose 0                        ' Close all windows.
MouseHide                            ' Hide the mouse cursor.
COLOR 15, 0                          ' Change colors back to original.
CLS                                  ' Clear the screen.
END
```

The code fragment opens two windows, each of a different color. Both contain some text about the features that are available in that window. Both also contain an OK command button that, when chosen, causes the selected window to close.

Both windows are movable. To move a window, move the mouse cursor to any of the dot-pattern (ASCII 176) characters adjacent to the title bar, or to the title bar itself. Press the left mouse button and drag a window by moving the mouse. Release the button when you are satisfied with the position.

Close a window by selecting the equal sign (=) in the upper-left corner of the window. Pressing Enter while either window is selected has the same effect as choosing the OK command button or selecting the window-close control character (=). Close a window by using **WindowClose**. **WindowClose** closes the window and any buttons or edit fields that may be open within the window.

You can move between the two windows and close either or both of them in any order. When both are closed, the program ends. When a window can be resized (the windows in the example cannot), a plus (+) character is displayed in the lower-right corner. Select the plus character and drag the mouse to resize the window.

The preceding information is intended to give you an overview of how the toolbox windows work. With this overview and the following detailed information about each procedure, you should be able to easily incorporate windows into your BASIC programs. The example code in UIDEMO.BAS demonstrates how most of the procedures are used.

A description of each procedure that comprises WINDOW.BAS follows.

Alert FUNCTION

Action Displays a window with one to three buttons that indicate choices for the user.

Syntax *variablename$* = **Alert**(*style%*, *text$*, *row1%*, *col1%*, *row2%*, *col2%*, *b1$*, *b2$*, *b3$*)

Remarks The **Alert** procedure returns an integer from 1 to 3, inclusive, indicating which button was selected by the user. Use **Alert** whenever you want to display a dialog box and force a user decision. The **Alert** procedure uses the following arguments:

Argument	Description
variablename$	Any BASIC variable name, including the name of a record variable or record element.

style%	An integer that indicates the style of the text to be displayed. The values of *style%* are:

Value	Significance
1	Truncated printing. If text is longer than the window, it is truncated. The text cursor is moved to the first character of the next line.
2	Character wrapping. If text is longer than the window, the text continues on the next line.
3	Text wrapping. Same as style 2 except that text is wrapped only at spaces.
4	Truncated centering. Text is centered on the current line. If text is too long, it is truncated.

text$	A string that contains the text to be displayed in the dialog box. You can force a new line in *text$* by using the vertical bar character (|). Use several together for more than one blank line.
row1%, *col1%*	An integer pair that specifies absolute screen row and column coordinates.
row2%, *col2%*	An integer pair that specifies the lower-right-corner coordinates of an area.
b1$	A string that contains the text for button 1.
b2$	A string that contains the text for button 2.
b3$	A string that contains the text for button 3.

If you use fewer than three buttons, use a null string ("") for the text of any unused buttons. All alert windows are modal; that is, they do not allow selection outside their boundaries. If you do not provide at least one button in your alert window, the **Alert** procedure will place a single OK button in the window for you.

You must ensure that the window you have described is large enough to contain all three buttons on the bottom row, that the window is at least three rows high, and that the window is large enough to contain *text$*. If the window is too small or any other problem occurs, the **Alert** procedure returns 0.

See Also **Dialog**, **ListBox**

Example See the DemoAlert procedure in the code example UIDEMO.BAS for a specific implementation.

BackgroundRefresh SUB

Action Used internally by other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **BackgroundRefresh** *handle%*

Remarks The **BackgroundRefresh** procedure restores a previously saved background from a global array to the area where a window was displayed. The argument *handle%* is an integer that indicates the number of the window whose background is being restored. This can be any number between 1 and the value declared in the constant MAXWINDOW.

If you want to restore a background, use the **PutBackground** procedure in GENERAL.BAS.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **WindowClose**, **WindowDo**, **WindowSetCurrent**

BackgroundSave SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **BackgroundSave** *handle%*

Remarks The **BackgroundSave** procedure saves the background (the area behind where a window is to be displayed) in a global array. The argument *handle%* is an integer that indicates the number of the window whose background is being saved. This can be any number between 1 and the value declared in the constant MAXWINDOW.

If you want to save a background, use the **GetBackground** procedure in GENERAL.BAS.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **WindowDo**, **WindowOpen**,**WindowSetCurrent**

ButtonClose SUB

Action Erases the specified button from the current window and deletes it from the global arrays.

Syntax **ButtonClose** *handle%*

Remarks The **ButtonClose** procedure is used to close individual buttons that have been opened with **ButtonOpen**. The argument *handle%* is an integer that indicates the number of the button being closed. This can be any number between 0 and the value declared in the constant MAXBUTTON, inclusive. If *handle%* is 0, all buttons in the current window are closed.

If the **WindowClose** procedure is used to close a window that contains buttons, you need not use this procedure, because all buttons associated with a closing window are closed automatically.

See Also **ButtonOpen**, **WindowClose**

ButtonInquire FUNCTION

Action Returns an integer that identifies the state of a specified button.

Syntax *variablename%* = **ButtonInquire**(*handle%*)

Remarks The **ButtonInquire** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that indicates the number of the button whose state is requested. This can be any number between 1 and the value declared in the constant MAXBUTTON, inclusive.

The **ButtonInquire** procedure is used when you need to determine the state of a type 1, 2, 3, 6, or 7 button, as specified in **ButtonOpen**.

See Also **ButtonSetState**

ButtonOpen SUB

Action Opens a button of the specified type, and places it in the current window at the window coordinates specified.

Syntax **ButtonOpen** *handle%*, *state%*, *text$*, *row1%*, *col1%*, *row2%*, *col2%*, *buttonType%*

Remarks Use **ButtonOpen** when you want to place any kind of buttons in open windows. The **ButtonOpen** procedure uses the following arguments:

Argument	Description
handle%	An integer that indicates the number of the button that is being opened. This can be any number between 1 and the value declared in the constant MAXBUTTON, inclusive.
state%	An integer that indicates the initial state of the button that is being opened. For each different button type and *state%* value, the following applies:

Type	Value	Significance
1	1	Normal.
	2	Default choice (brackets highlighted).
	3	Chosen (highlighted in reverse video).
2 & 3	1	Normal.
	2	Selected (checked).
4	0	Does not apply to type 4 (area button).
5		Reserved for future use.
6 & 7	*n*	Indicates the initial position of the scroll bar's position indicator. It can be between 1 and the maximum position.

Argument	Description
text$	A string that contains the text to be displayed in the button. Does not apply to button types 4, 6, or 7.
row1%, *col1%*	An integer pair that describes the upper-left corner of an area relative to the upper-left corner of the current window, not the screen.

row2%, col2%	An integer pair that specifies the lower-right corner of an area, relative to the upper-left corner of the current window, not the screen. The coordinates *row2%* and *col2%* apply only to button types 4, 6, and 7. Any values in these arguments are ignored by other button types; however, arguments must be in the argument list, even if their value is 0.
buttonType%	An integer that indicates the type of button to be opened. The following is a list of valid button types:

Value	Description
1	Command button
2	Check box
3	Option button
4	Area button
5	Reserved for future use
6	Vertical scroll bar
7	Horizontal scroll bar

Note that the positioning coordinates are relative to the current window, not absolute from the top-left corner of the screen. Within each window, each button has a unique *handle%* value. Because buttons are local to specific windows it is permissible to have more than one button with the same *handle%* value, provided they are in different windows.

See Also **ButtonClose**

ButtonSetState SUB

Action Sets the state of the specified button and redraws it.

Syntax **ButtonSetState** *handle%*, *state%*

Remarks The **ButtonSetState** procedure operates on buttons in the current window. Use **ButtonSetState** to change button state in response to user selection of different buttons. The **ButtonSetState** procedure uses the following arguments:

Argument	Description
handle%	An integer that indicates the number of the button whose state is being set. This can be any number between 1 and the value declared in the constant MAXBUTTON, inclusive.

state% An integer that indicates the state of the button. For each different button type and *state%* value, the following applies:

Type	Value	Significance
1	1	Normal.
	2	Default choice (brackets highlighted).
	3	Chosen (highlighted in reverse video).
2 & 3	1	Normal.
	2	Selected (checked).
4	0	Does not apply to type 4.
5		Reserved for future use.
6 & 7	*n*	Sets the initial position of the scroll bar's position indicator. It can be between 1 and the maximum position.

See Also **ButtonInquire**

ButtonShow SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **ButtonShow** *handle%*

Remarks The **ButtonShow** procedure draws a button on the screen, and is used automatically by WINDOW.BAS. Normally, you never need to call this procedure directly since it is automatically called when needed. The argument *handle%* is an integer that indicates the number of the button that is being opened. This can be any number between 1 and the value declared in the constant MAXBUTTON, inclusive.

Warning Do not alter this procedure unless you are customizing the User Interface toolbox and know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **ButtonOpen**, **ButtonSetState**, **ButtonToggle**

ButtonToggle SUB

Action Toggles a button state between state 1 (normal) and state 2 (selected).

Syntax **ButtonToggle** *handle%*

Remarks Use **ButtonToggle** to toggle button state in response to user selection of different buttons. The argument *handle%* is an integer that indicates the number of the button whose state is being toggled. This can be any number between 1 and the value declared in the constant MAXBUTTON. **ButtonToggle** applies only to type 1, 2, or 3 buttons.

See Also **ButtonSetState**

Dialog FUNCTION

Action Returns an integer whose value indicates what type of button, edit-field, or window event occurred within the **WindowDo** procedure.

Syntax *variablename%* = **Dialog**(*op%*)

Remarks The **Dialog** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
op%	An integer that defines a specific operation that **Dialog** is to evaluate. The following table lists the possible values of *op%* and the information that is returned:

op%	Return	Event/significance
0	0	No event took place.
	1	A button was pressed. Use `Dialog(1)` to determine which button.
	2	An edit field was selected. Use `Dialog(2)` to determine which field.
	3	A window other than the current window was selected. Use `Dialog(3)` to determine which window.

	4	Current window's close box was selected.
	5	Current window needs to be refreshed because it was resized.
	6	Enter key was pressed.
	7	Tab key was pressed.
	8	Shift+Tab was pressed.
	9	Esc was pressed.
	10	Up Arrow key was pressed.
	11	Down Arrow key was pressed.
	12	Left Arrow key was pressed.
	13	Right Arrow key was pressed.
	14	Spacebar was pressed.
	15	Current window was moved.
	16	Home key was pressed.
	17	End key was pressed.
	18	PgUp key was pressed.
	19	PgDn key was pressed.
	20	Menu item was chosen.
1	*n*	Returns the number of the button just pressed. May be from 1 to the value declared in MAXBUTTON, inclusive.
2	*n*	Returns the number of the edit field just selected. May be from 1 to the value declared in MAXEDITFIELD, inclusive.
3	*n*	Returns the number of the window just selected. May be from 1 to the value declared in MAXWINDOW, inclusive.
4 – 15		Reserved for future use.
17	*n*	Returns the row of the cursor within the field of button type 4. (See **ButtonOpen**.) May be from 1 to the value declared in MAXROW inclusive.
18	*n*	Returns the column of the cursor within the field of button type 4. (See **ButtonOpen**.) May be from 1 to the value declared in MAXCOL, inclusive.

19	–2	Selected Down or Right direction key on scroll bar (button type 6 or 7).
	–1	Selected Up or Left direction key on scroll bar (button type 6 or 7)
	n	Selected a position on grayed area of scroll bar. May be from 1 to the maximum number of positions on the scroll bar.
20		Reserved for future use.

Using an *op%* value of 0, you can determine *what* happened; using the other arguments, you can determine *where* it happened.

See Also **Alert**, **ListBox**

Example See the individual procedures in the code example UIDEMO.BAS for examples of specific usage.

EditFieldClose SUB

Action Erases the specified edit field from the current window and deletes it from the global arrays.

Syntax **EditFieldClose** *handle%*

Remarks Use **EditFieldClose** when you want to erase an edit field from a window rather than close the window. The argument *handle%* is an integer that indicates the number of the edit field being closed. This can be any number between 0 and the value declared in the constant MAXEDITFIELD, inclusive. If *handle%* is 0, all edit fields in the current window are closed. If the **WindowClose** procedure is used to close a window that contains edit fields, you need not use this procedure since all edit fields associated with a closing window are automatically closed.

See Also **EditFieldOpen**, **WindowClose**

EditFieldInquire FUNCTION

Action Returns the string associated with the specified edit field.

Syntax *variablename$* = **EditFieldInquire**(*handle%*)

Remarks Use **EditFieldInquire** when you need to return the edit string associated with a particular edit field. The **EditFieldInquire** procedure uses the following arguments:

Argument	Description
variablename$	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that indicates the number of the edit field whose edit string is requested. This can be any number between 1 and the value declared in the constant MAXEDITFIELD, inclusive.

See Also **EditFieldOpen**

Example See the DemoDialog procedure in the code example UIDEMO.BAS for a specific implementation.

EditFieldOpen SUB

Action Opens an edit field and places it in the current window at the window coordinates specified.

Syntax **EditFieldOpen** *handle%, text$, row%, col%, fore%, back%, visLength%, maxLength%*

Remarks Use the **EditFieldOpen** procedure anytime you want to open an edit field in the current window. The **EditFieldOpen** procedure uses the following arguments:

Argument	Description
handle%	An integer that indicates the number of the edit field that is being opened. This can be any number between 1 and the value declared in the constant MAXEDITFIELD, inclusive.
text$	A string that contains the text that is to be initially placed in the edit field for editing.

row%, col%	An integer pair that describes the upper-left corner of an area, relative to the upper-left corner of the current window, not the screen.
fore%	An integer that defines the edit field foreground color (0–15)
back%	An integer that defines the edit field background color (0–7)
visLength%	An integer that indicates the length of the edit field that is visible on the screen.
maxLength%	An integer that indicates the maximum length of the edit field, up to a maximum of 255 characters.

Note that the positioning coordinates are relative to the current window, not absolute from the top left of the screen. The edit field foreground and background colors may be different from the rest of the window. If the maximum length of the edit field is greater than the visible length of the edit field, the field will scroll left and right, as needed, during editing.

See Also **EditFieldClose**

Example See the DemoDialog procedure in the code example UIDEMO.BAS for specific usage.

FindButton FUNCTION

Action Used internally by the button procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax *variablename%* = **FindButton**(*handle%*)

Remarks The **FindButton** procedure returns an index into the global array where a button is stored. Each button is uniquely described by the window in which it exists, and a button handle. **FindButton** assumes the current window.

The **FindButton** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that indicates the number of the button whose index is requested. This can be any number between 1 and the value declared in the constant MAXBUTTON, inclusive.

Warning Do not alter this procedure unless you are customizing the User Interface toolbox and you know how the alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **ButtonClose**, **ButtonInquire**, **ButtonSetState**, **ButtonShow**, **ButtonToggle**, **MaxScrollLength**, **WindowDo**

FindEditField FUNCTION

Action Used internally by WINDOW.BAS. Do not use or alter this procedure.

Syntax *variablename%* = **FindEditField**(*handle%*)

Remarks The **FindEditField** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that indicates the number of the edit field whose index is requested. This can be any number between 1 and the number declared in the constant MAXEDITFIELD, inclusive.

The **FindEditField** procedure returns an index into the global array where an edit field is stored. Each edit field is uniquely described by the window in which it exists, and an edit field handle. **FindEditField** assumes the current window.

Warning Do not alter this procedure unless you are customizing the User Interface toolbox and you know how the alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **EditFieldClose**, **EditFieldInquire**, **WindowDo**

ListBox FUNCTION

Action Displays a window containing a list box, a scroll bar, an OK button, and a Cancel button.

Syntax *variablename%* = **ListBox** (*text$*( ), *maxRec%*)

Remarks

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
text$( )	A string array that contains the text items to be displayed in the list box.
maxRec%	An integer that defines the maximum number of entries in the list box.

With the **ListBox** procedure, you can use the mouse or keyboard to select any single item from the text in a list box. To accept the current choice, select OK. Use Cancel to reject all choices and close the list box. The **ListBox** procedure returns an integer that is the number of the item selected from the text in the list box when OK is selected, or 0 if Cancel is selected.

Use **ListBox** when you have a list of things to choose from that may be longer than the available screen space.

See Also **Alert, Dialog**

Example See the DemoListBox procedure in the code example UIDEMO.BAS for a specific usage.

MaxScrollLength FUNCTION

Action Returns an integer that is the maximum number of positions in the specified scroll bar.

Syntax *variablename%* = **MaxScrollLength**(*handle%*)

Remarks The **MaxScrollLength** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that specifies the scroll bar for which the maximum number of positions is requested. This is the number that is used to identify the scroll bar when it is opened using **ButtonOpen**. This can be any number between 1 and the value declared in the constant MAXBUTTON, inclusive.

The **MaxScrollLength** procedure is used to determine how many positions your scroll bar has. Use this procedure to increment or decrement the cursor position in response to user selection of the direction arrows to move the cursor.

See Also **ListBox**

Example See the DemoScrollBar procedure in the code example UIDEMO.BAS for a specific usage.

WhichWindow FUNCTION

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax *variablename%* = **WhichWindow**(*row%*, *col%*)

Remarks The **WhichWindow** procedure returns an integer that is the window number for the location specified on the screen. Overlapping windows are taken into account. The **WhichWindow** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
row%, col%	An integer pair that describes a particular point on the screen.

Warning Do not alter this procedure unless you're customizing the User Interface toolbox and know how the alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **WindowDo**

WindowBorder FUNCTION

Action Returns a 14-character string that describes the border of the window.

Syntax *border$* = **WindowBorder** (*windowType%*)

Remarks The argument *windowType%* is an integer that describes a particular window type.

The border of the window actually determines what characteristics a window will have. Each character position in the returned string is representative of a particular component of the border. Some positions in the border have added significance, in that special characters in those positions implement certain features. The following list indicates the significance of each *border$* character position and the features implemented by special characters:

Position	Character defined	Special effect character
1	Upper-left corner	Equal sign (=) makes window closable.
2	Top line	Dot-pattern character (ASCII 176) makes window movable.
3	Upper-right corner	
4	Left-side line	
5	Middle fill character	
6	Right-side line	
7	Lower-left corner	
8	Bottom line	
9	Lower-right corner	Plus sign (+) makes window resizable.
10	Left-side intersection	
11	Middle line	
12	Right-side intersection	
13	Shadow flag	"S" in this position puts a 3D shadow on window. All predefined window types have shadows.
14	Title flag	"T" in this position puts a title bar across the top of the window if the length of *title$* in **WindowOpen** is greater than 0.

Twenty-four basic predefined window types are provided in WINDOW.BAS. These combinations include all of the special features (movable, closable, resizable, shadow, and title) with three different borders. You can have no border, a single-line border, or a double-line border. All available options can be specified with arguments to the **WindowOpen** procedure.

If you want to create other custom window types, modify the **WindowBorder** procedure in WINDOW.BAS to add more window types. By observing the conventions in the list above, you can create windows that have the features and any special borders you want. To create custom window types, modify this procedure and recompile it into your Quick library and .LIB file.

Note The **WindowBorder** procedure is used internally by WINDOW.BAS and under normal circumstances will not be called directly by your program. You may wish to modify and recompile this procedure to extend or alter its functionality.

See Also **ButtonOpen, EditFieldOpen, WindowClose, WindowDo, WindowOpen, WindowPrintTitle, WindowSetCurrent**

WindowBox SUB

Action Draws a box with a single-line border, within the current window at the coordinates specified.

Syntax **WindowBox** *boxRow1%, boxCol1%, boxRow2%, boxCol2%*

Remarks The **WindowBox** procedure uses the following arguments:

Argument	Description
boxRow1%, boxCol%	An integer pair that describes the upper-left corner of an area relative to the upper-left corner of the current window.
boxRow2%, boxCol2%	An integer pair that describes the lower-right corner of an area relative to the upper-left corner of the current window.

The **WindowBox** procedure is used by the **ListBox** procedure to create a single-line box within the window. **WindowBox** can be used to box any window area as desired.

See Also ListBox

WindowClose SUB

Action Closes a specified window, closing all buttons and edit fields associated with that window.

Syntax **WindowClose** *handle%*

Remarks The argument *handle%* is an integer that indicates the number of the window being closed. This can be any number between 0 and the value declared in the constant MAXWINDOW, inclusive. When you close the current window, the top window becomes the current window. Any button and edit fields associated with the closing window are also closed. If *handle%* is 0, all windows are closed.

See Also **WindowOpen**

Example See the DemoWindow procedure in the code example UIDEMO.BAS for a specific usage that closes all open windows.

WindowCls SUB

Action Clears the current window.

Syntax **WindowCls**

Remarks The **WindowCls** procedure is used whenever you want to erase the text contained in a window. **WindowCls** clears the current window using the window background color (not the text background color) defined when the window was first opened with **WindowOpen.**

See Also **WindowColor, WindowOpen, WindowScroll**

Example See the DemoResize procedure in the code example UIDEMO.BAS for a specific implementation.

WindowColor SUB

Action Reassigns the values of the text color variables for the current window.

Syntax **WindowColor** *textFore%*, *textBack%*

Remarks The **WindowColor** procedure uses the following arguments:

Argument	Description
textFore%	An integer that defines foreground color (0–15) of text.
textBack%	An integer that defines background color (0–7) of text.

Use **WindowColor** whenever you want to change the color of the text being printed in a window. Any printing after using **WindowColor** is done in the newly defined colors.

See Also WindowOpen

WindowCols FUNCTION

Action Returns an integer that is the number of interior columns in the current window.

Syntax *variablename%* = **WindowCols** (*handle%*)

Remarks The **WindowCols** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that indicates the number of the window whose column count is requested. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

Use **WindowCols** when you want to determine the number of columns in a window, especially when dealing with resizable windows. These "interior" columns are those where text can be displayed.

See Also WindowRows

WindowCurrent FUNCTION

Action Returns an integer that is the handle number of the current window.

Syntax *variablename%* = **WindowCurrent**

Remarks The argument *variablename%* is any BASIC variable name, including the name of a record variable or record element. Use **WindowCurrent** anywhere you need the number of the current window as an argument for other window procedures (for example, **WindowClose** and **WindowCurrent**).

See Also **WindowSetCurrent**

Example See the DemoWindow procedure in the code example UIDEMO.BAS for a specific usage example.

WindowDo SUB

Action Takes control of the program and waits for a button, edit-field, or window event to occur in an open window.

Syntax **WindowDo** *currButton%*, *currEdit%*

Remarks The **WindowDo** procedure uses the following arguments:

Argument	Description
currButton%	An integer that indicates the number of the current button. The current button is that button where the text cursor is located at the time of the call to **WindowDo**.
currEdit%	An integer that indicates the number of the current edit field. The current edit field is the edit field where the text cursor is located at the time of the call to **WindowDo**. If it is 0, it is assumed that tne text cursor is not currently in an edit field. If it is nonzero, **WindowDo** lets the user edit the current edit field.

Only buttons and edit fields in the current window are active. If no buttons or edit fields are used in the window, use 0 for both arguments. Once an event takes place, use `Dialog(0)` to determine which event occurred.

Program execution continues after an event occurs.

See Also **Dialog**

Example See the DemoDialog, DemoDialogEZ, DemoResize, DemoScrollBar, and DemoWindow procedures in the code example UIDEMO.BAS for specific usage examples.

WindowInit SUB

Action Initializes the global window, button, and edit-field arrays.

Syntax **WindowInit**

Remarks The **WindowInit** procedure is used at or near the beginning of any program that uses the window procedures provided by the User Interface toolbox. You must call **WindowInit** before any other window procedures are used. You can execute **WindowInit** again anytime you need to reset the windowing environment.

See Also **ButtonOpen**, **EditFieldOpen**, **WindowOpen**

WindowLine SUB

Action Draws a horizontal line across the current window at the row specified.

Syntax **WindowLine** *row%*

Remarks The argument *row%* is an integer that describes a row relative to the top of the current window. The **WindowLine** procedure is used in dialog boxes to separate parts. **WindowLine** is used by the **Alert** procedure to draw a horizontal line above the buttons.

See Also **WindowPrint**

Example See the DemoDialogEZ procedure in the code example UIDEMO.BAS for a specific implementation.

WindowLocate SUB

Action Sets the row and column of the window text cursor in a window.

Syntax **WindowLocate** *row%*, *col%*

Remarks The coordinates *row%* and *col%* are integers that describe a print position relative to the upper-left corner of the current window. The **WindowLocate** procedure is used, once a window has been opened, to position the window text cursor to the starting point of a line of text. This is the position where the next character will be placed when a **WindowPrint** operation occurs.

See Also **WindowPrint**

Example See the DemoDialog procedure in the code example UIDEMO.BAS for a specific implementation.

WindowNext FUNCTION

Action Returns an integer that is the handle number of the next available window.

Syntax *variablename%* = **WindowNext**

Remarks The **WindowNext** procedure is used in situations where a window is to be opened, but the opening routine has no information about other windows that may already be open. The argument *variablename%* is any BASIC variable name, including the name of a record variable or record element.

See Also **WindowCurrent, WindowOpen**

WindowOpen SUB

Action Defines and opens a window.

Syntax **WindowOpen** *handle%, row1%, col1%, row2%, col2%, textFore%, textBack%, fore%, back%, highlight%, movewin%, closewin%, sizewin%, modalwin%, borderchar%, title$*

Remarks

The **WindowOpen** procedure uses the following arguments:

Argument	Description
handle%	An integer that indicates the number of the window that is being opened. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.
row1%, col1%	An integer pair that specifies absolute screen row and column coordinates.
row2%, col2%	An integer pair that specifies the lower-right-corner coordinates of an area.
textFore%	An integer that defines text foreground color (0–15).
textBack%	An integer that defines text background color (0–7).
fore%	An integer that defines window foreground color (0–15).
back%	An integer that defines window background color (0–7).
highlight%	An integer that defines color (0–15) of highlighted buttons.
movewin%	TRUE indicates a movable window; FALSE indicates a stationary window.
closewin%	TRUE indicates a window that can be closed; FALSE indicates a window that can't be closed.
sizewin%	TRUE indicates a resizable window; FALSE indicates a window that can't be resized.
modalwin%	TRUE indicates a modal window; FALSE indicates a nonmodal window. When a window is modal, any attempt to select outside the window is unsuccessful and results in a beep.
borderchar%	0 indicates no border; 1 indicates a single-line border; 2 indicates a double-line border.
title$	A string that is the title of the window. If *title$* is a null string (""), no title is displayed.

The **WindowOpen** procedure is used to open windows anywhere on the screen. Twenty-four basic window types are predefined in WINDOW.BAS; however, the fact that any of the basic types can be created as either a standard or modal window increases the available number to 48.

See Also

WindowBorder, WindowClose, WindowNext

Example

See the DemoWindow procedure in the code example UIDEMO.BAS for a specific implementation.

WindowPrint SUB

Action Prints text, in the style specified, in the current window at the position established by **WindowLocate**.

Syntax **WindowPrint** *style%*, *text$*

Remarks The **WindowPrint** procedure uses the following arguments:

Argument	Description
style%	An integer that indicates the style of the text to be displayed. Possible values for *style%* and their significance are described as follows:

Value	Significance
1	Truncated printing. If text is longer than the window, it is truncated. The text cursor is moved to the first character of the next line. The window is scrolled, if necessary.
2	Character wrapping. If text is longer than the window, the text continues on the next line, scrolling if necessary. The text cursor is moved to the first character on the next line, scrolling if necessary.
3	Text wrapping. Same as style 2 except that text is wrapped only at spaces.
4	Truncated centering. Text is centered on the current line. If text is too long, it is truncated.
–1	Same as style 1 except, after printing, text cursor is positioned immediately following the last character printed.
–2	Same as style 2 except, after printing, text cursor is positioned immediately following the last character printed.
–3	Same as style 3 except, after printing, text cursor is positioned immediately following the last character printed.

Argument	Description
text$	A string that contains the text to be displayed in the window.

Use **WindowPrint** whenever you want to display text in a window.

See Also WindowLocate

Example See the DemoWindow procedure in the code example UIDEMO.BAS for a specific implementation.

WindowPrintTitle SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **WindowPrintTitle**

Remarks The **WindowPrintTitle** procedure prints a window's title bar.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also WindowDo

WindowRefresh SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **WindowRefresh** *handle%*

Remarks The **WindowRefresh** procedure restores a window to the screen from a global array. The *handle%* is an integer that indicates the number of the window being restored. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

To restore part of the screen, use the **PutBackground** procedure in GENERAL.BAS.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also WindowDo

WindowRows FUNCTION

Action Returns an integer that is the number of interior rows in the current window.

Syntax *variablename%* = **WindowRows** (*handle%*)

Remarks The **WindowRows** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
handle%	An integer that indicates the number of the window whose row count is requested. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

Use **WindowRows** when you need to determine the number of interior rows in a window. Interior rows are those where text can be displayed. The **WindowRows** procedure can be used when resizing windows that contain text, to ensure that you don't print more text than there are lines available to print on.

See Also **WindowCols**

Example See the DemoWindow procedure in the code example UIDEMO.BAS for a specific implementation.

WindowSave SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **WindowSave** *handle%*

Remarks The **WindowSave** procedure saves a window into a global array. If you want to save part of the screen, use the **GetBackground** procedure in GENERAL.BAS. The argument *handle%* is an integer that indicates the number of the window being saved. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **WindowDo**

WindowScroll SUB

Action Scrolls text in the current window by the number of lines specified.

Syntax **WindowScroll** *lines%*

Remarks The **WindowScroll** procedure is used in the **WindowPrint** procedure of the User Interface toolbox. The argument *lines%* is an integer that defines the number of lines to scroll. If *lines%* is greater than 0, the window scrolls up; if *lines%* is less than 0, the window scrolls down. If *lines%* equals 0, the window is cleared.

See Also **WindowCls, WindowPrint**

WindowSetCurrent SUB

Action Makes the specified window the current window, placing it on top (in front) of any other windows that may be open.

Syntax **WindowSetCurrent** *handle%*

Remarks The **WindowSetCurrent** procedure is used to change to a new window. Only the current window is displayed with a shadow effect. The argument *handle%* is an integer that indicates the number of the window to make current. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

See Also **WindowCurrent**

Example See the DemoWindow procedure in the code example UIDEMO.BAS for a specific implementation.

WindowShadowRefresh SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **WindowShadowRefresh** *handle%*

Remarks The **WindowShadowRefresh** procedure restores the background behind a shadow from a global array. If you want to restore part of the screen, use the **PutBackground** procedure in GENERAL.BAS. The argument *handle%* is an integer that indicates the number of the window whose shadow background is being restored. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **WindowDo**

WindowShadowSave SUB

Action Used internally by the other procedures in WINDOW.BAS. Do not use or alter this procedure.

Syntax **WindowShadowSave** *handle%*

Remarks The **WindowShadowSave** procedure saves the background (the area behind where a window's shadow is to be displayed) into a global array. If you want to save part of the screen, use the **GetBackground** procedure in GENERAL.BAS. The argument *handle%* is an integer that indicates the number of the window whose shadow background is being saved. This can be any number between 1 and the value declared in the constant MAXWINDOW, inclusive.

Warning Do not use or alter this procedure unless you are customizing the User Interface toolbox and you know how the use or alteration of this procedure will affect the operation of all other procedures in the toolbox.

See Also **WindowDo**

MOUSE.BAS

The MOUSE.BAS source-code file supports the Microsoft Mouse and other pointing devices that have a 100 percent Microsoft-Mouse-compatible driver. Although the procedures can be used by themselves, they were designed as an integral part of the User Interface toolbox.

If you use the procedures in MOUSE.BAS, you must include GENERAL.BI and MOUSE.BI in your program so you will have the proper declarations and definitions. The header file MOUSE.BI includes procedure declarations for the MOUSE.BAS file.

A description of the procedures that comprise MOUSE.BAS follows.

MouseBorder SUB

Action Establishes the limits of mouse movement on the screen.

Syntax **MouseBorder** *row1%, col1%, row2%, col2%*

Remarks The **MouseBorder** procedure uses the following arguments:

Argument	Description
row1%, col1%	An integer pair that specifies absolute screen row and column coordinates.
row2%, col2%	An integer pair that specifies the lower-right corner coordinates of an area.

MouseBorder is used in WINDOW.BAS to establish the limits of mouse movement when windows are moved or resized. Mouse movement is permitted only within the boundary described by the parameters of **MouseBorder**.

See Also **WindowDo, MouseInit**

MouseDriver SUB

Action Calls Interrupt 51 (33H) and passes the parameters to the proper registers.

Syntax **MouseDriver** *M0%, M1%, M2%, M3%*

Remarks The **MouseDriver** procedure uses the following arguments:

Argument	Description
M0%	An integer that is placed into the AX register.
M1%	An integer that is placed into the BX register.
M2%	An integer that is placed into the CX register.
M3%	An integer that is placed into the DX register.

Interrupt 51 provides DOS mouse services. Refer to the Microsoft Mouse *Programmer's Guide* for detailed information about mouse servicing routines. **MouseDriver** is used in MOUSE.BAS (all procedures) where access to operating system mouse routines is required.

See Also **MouseBorder**, **MouseHide**, **MouseInit**, **MousePoll**, **MouseShow**

MouseHide SUB

Action Decrements a cursor flag in the mouse driver, causing the mouse cursor to be hidden.

Syntax **MouseHide**

Remarks Use this procedure before each time you print something on the screen, to prevent the mouse cursor from being overwritten. Once you've written to the screen, be sure to use a corresponding **MouseShow** procedure to turn the mouse cursor back on; otherwise you will be unable to see your mouse cursor.

When the internal cursor flag is less than 0, the mouse cursor is hidden.; when it is 0, the mouse cursor is displayed. It is never greater than 0.

MouseHide is used throughout the User Interface toolbox anytime something is displayed on the screen.

Note The cursor flag is part of the mouse driver, not part of these mouse procedures or the User Interface toolbox.

See Also **MouseDriver**, **MouseShow**

MouseInit SUB

Action Initializes the mouse-driver-servicing routines.

Syntax **MouseInit**

Remarks **MouseInit** is used at or near the beginning of any program that uses the procedures provided by the mouse procedures of the User Interface toolbox. **MouseInit** must be called before any other mouse procedures are used.

The **MouseInit** procedure sets the cursor flag in the mouse driver to –1. When the internal cursor flag is less than 0, the mouse cursor is hidden; when it is 0, the mouse cursor is displayed. It is never greater than 0.

Note The cursor flag is part of the mouse driver, not part of these mouse procedures or the User Interface toolbox.

See Also **MenuInit**, **MouseDriver**

MousePoll SUB

Action Polls the mouse driver.

Syntax **MousePoll** *row%*, *col%*, *lbutton%*, *rbutton%*

Remarks The **MousePoll** procedure uses the following arguments:

Argument	Description
row%	A variable that contains the row coordinate for the mouse cursor.
col%	A variable that contains the column coordinate for the mouse cursor.
lbutton%	A variable that contains the status of the left mouse button.
rbutton%	A variable that contains the status of the right mouse button.

MousePoll places the row and column position of the mouse cursor into the *row%* and *col%* variables. The **MousePoll** procedure also determines the status of the left and right mouse buttons, and places the status (TRUE if pressed or FALSE if not) into the *lbutton%* and *rbutton%* variables, as appropriate.

MousePoll is used throughout the User Interface toolbox whenever the location of the mouse cursor or the status of the mouse buttons is required.

See Also **MouseDriver**

MouseShow SUB

Action Increments a cursor flag in the mouse driver that causes the mouse cursor to be displayed.

Syntax **MouseShow**

Remarks The **MouseShow** procedure is used each time you print something to the screen; it rewrites the mouse cursor to the screen. You should use a **MouseShow** procedure for each **MouseHide** you use in your program.

When the internal cursor flag is less than 0, the mouse cursor is hidden. When it is 0, the mouse cursor is displayed. It is never greater than 0. If the internal cursor flag is already 0, this procedure has no effect. **MouseShow** is used throughout the User Interface toolbox whenever anything is displayed on the screen.

Note The cursor flag is part of the mouse driver, not part of these mouse procedures or the User Interface toolbox.

See Also **MouseDriver**, **MouseHide**

GENERAL.BAS

The GENERAL.BAS source-code file contains a number of general purpose character-based routines that are used in both the MENU.BAS and WINDOW.BAS files. While these files can be used by themselves, they are intended to support the other procedures in the User Interface toolbox. If you use any of the procedures in GENERAL.BAS, you must include the header file, GENERAL.BI.

A description of the procedures that comprise GENERAL.BAS follows.

AltToASCII$ FUNCTION

Action

Decodes the extended key codes associated with the Alt key, and returns only the individual ASCII character.

Syntax

*variablename$* = **AltToASCII$** (*kbd$*)

Remarks

The **AltToASCII$** procedure uses the following arguments:

Argument	Description
variablename$	Any BASIC variable name, including the name of a record variable or record element.
kbd$	A string that contains a character entered at the keyboard.

AltToASCII$ is used in the procedures in MENU.BAS to identify access keys that have been pressed for menu-item selection (Alt + 0–9, A–Z, – and =).

For instance, pressing the Alt key with the "A" key places two scan-code values into the keyboard buffer, CHR$(0)+CHR$(30). **AltToASCII$** strips off CHR$(0) and returns "A," which is the letter represented by scan code 30.

The keyboard can be polled with either the BASIC **INKEY$** procedure or the **MenuInkey$** procedure (preferred) included in the MENU.BAS file in the User Interface toolbox.

See Also

MenuDo

AttrBox SUB

Action Changes the color of the characters within a box.

Syntax **AttrBox** *row1%, col1%, row2%, col2%, attr%*

Remarks The **AttrBox** procedure uses the following arguments:

Argument	Description
row1%, col1%	An integer pair that specifies absolute screen row and column coordinates.
row2%, col2%	An integer pair that specifies the lower-right corner coordinates of an area.
attr%	An integer that defines the color and intensity attributes for your particular display adapter. The number is derived by adding the product of 16 times the value of the background color (0–7) to the value of the foreground color (0–15). See the entries for the **SCREEN** and **PALETTE** statement for more information.

AttrBox changes the color attribute of characters within a box, described by column and row coordinates, to the attribute contained in the argument *attr%*. The characters within the described area are unaffected, except for changes in color.

AttrBox is used in MENU.BAS to provide shadows for the menus, and in WINDOW.BAS to draw the shadow on newly displayed windows.

See Also **MenuDo, WindowShadowSave**

Box SUB

Action Draws a box around a defined area, using the foreground and background colors specified.

Syntax **Box** *row1%, col1%, row2%, col2%, fore%, back%, border$, fillflag%*

Remarks

The **Box** procedure uses the following arguments:

Argument	Description
row1%, *col1%*	An integer pair that specifies absolute screen row and column coordinates.
row2%, *col2%*	An integer pair that specifies the lower-right corner coordinates of an area.
fore%	An integer that defines the foreground color (0–7).
back%	An integer that defines the background color (0–15).
border$	Nine-character string that defines the characters that are used to create the box. Each character position is significant because it defines a particular part of the box. Characters are defined as follows:

Position	Character described
1	Upper-left-corner character
2	Top-line character
3	Upper-right-corner character
4	Left-side line character
5	Fill character for middle area
6	Right-side line character
7	Lower-left-corner character
8	Bottom-line character
9	Lower-right-corner character

Argument	Description
fillflag%	TRUE (–1) if the interior of the box is to be cleared; FALSE (0) if it is not.

The **Box** procedure is used in WINDOW.BAS to draw boxes on the screen.

The individual character entries in the argument *border$* define which characters are used to draw the box. You can use any characters in either the standard ASCII or extended character set. Normally, box characters in the range CHR$(179) through CHR$(218) are used. (The figure on the following page shows the box characters and their corresponding ASCII codes.)

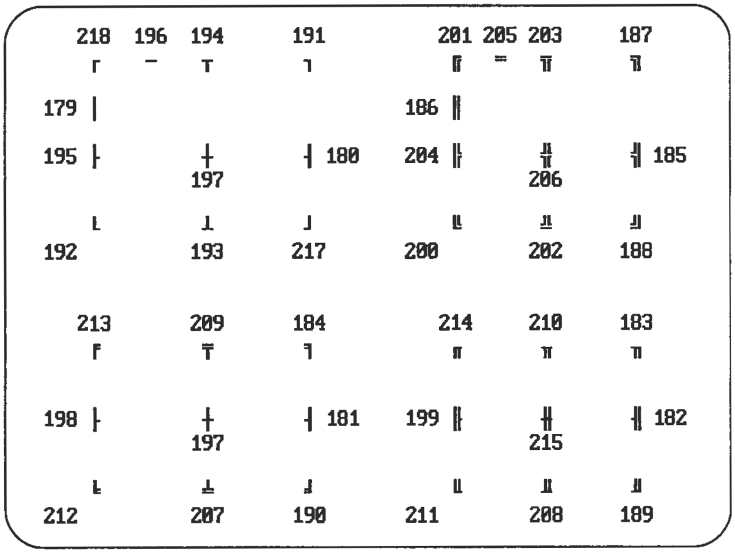

Figure 3.1 Box border characters and their corresponding ASCII codes

Box characters can be entered by pressing the Alt key and the appropriate numbers on the keypad. The function of each of the nine character positions in the argument *border$* is shown in the table above.

If fewer than nine characters are included in *border$*, a default single-line box is drawn; if more than nine characters are included, only the first nine are used.

See Also **WindowBox, WindowDo, WindowOpen**

GetBackground SUB

Action Saves an area of the screen into a named buffer.

Syntax **GetBackground** *row1%, col1%, row2%, col2%, buffer$*

Remarks The **GetBackground** procedure uses the following arguments:

Argument	Description
row1%, col1%	An integer pair that specifies absolute screen row and column coordinates.
row2%, col2%	An integer pair that specify the lower-right corner coordinates of an area.
buffer$	Name of the buffer variable where the defined area is to be stored.

GetBackground is used in MENU.BAS to store in a buffer the area (for example, the background where a window is to be displayed) that a menu is to occupy, before the menu is displayed. **GetBackground** is also used in WINDOW.BAS to store in a buffer the area that a window is to occupy, before the window is displayed. When used with MENU.BAS or WINDOW.BAS, a buffer is created for each menu or window, up to the maximum allowable (normally 10 of each).

There must be a buffer for each different area that is to be stored. The **GetBackground** procedure creates enough space in each buffer to hold the defined screen area. It then uses an assembly-language routine (GetCopyBox) to perform the actual screen-save operation.

See Also **MenuDo, PutBackground, WindowDo, WindowSave, WindowShadowSave**

GetShiftState FUNCTION

Action Returns the status of one of eight shift states.

Syntax *variablename%* = **GetShiftState** (*bit%*)

Remarks The **GetShiftState** procedure uses the following arguments:

Argument	Description
variablename%	Any BASIC variable name, including the name of a record variable or record element.
bit%	An integer between 0 and 7, inclusive, that defines which key status is requested.

The returned value is TRUE if the state of the key is pressed (Alt, Ctrl, Shift) or On if the key operates as a toggle (Scroll Lock, NumLock, Caps Lock, Ins). If the key is not pressed or On, FALSE (0) is returned. The values for *bit%* and the keys represented are described as follows:

Value	Key status (return value)
0	Right Shift pressed (TRUE)/not pressed (FALSE)
1	Left Shift pressed (TRUE)/not pressed (FALSE)
2	Ctrl pressed (TRUE)/not pressed (FALSE)
3	Alt pressed (TRUE)/not pressed (FALSE)
4	Scroll Lock pressed (TRUE)/not pressed (FALSE)
5	NumLock toggle on (TRUE)/off (FALSE)
6	Caps Lock toggle on (TRUE)/off (FALSE)
7	Ins toggle on (TRUE)/off (FALSE)

The **GetShiftState** procedure is used in MENU.BAS to determine whether the Alt key has been pressed, so that user input can be properly processed to select the desired menu item. **GetShiftState** is also used in WINDOW.BAS when editing edit fields.

See Also **MenuDo, MenuEvent, WindowDo**

PutBackground SUB

Action Restores the background from the named buffer to the row and column coordinates specified.

Syntax **PutBackground** *row%, col%, buffer$*

Remarks The **PutBackground** procedure uses the following arguments:

Argument	Description
row%, col%	An integer pair that specifies absolute screen row and column coordinates.
buffer$	Name of the buffer variable where the defined area was stored by the **GetBackground** procedure.

PutBackground is used in MENU.BAS to restore an area on the screen from a buffer, after closing a menu. **PutBackground** is also used in WINDOW.BAS to restore an area on the screen from a buffer, after closing a window. The complementary action, that of storing the background in a buffer before a menu or window is displayed, is performed by the **GetBackground** procedure.

Normally, a background is restored to the coordinates from which it was previously saved before a menu or window was displayed. The **PutBackground** procedure checks the screen boundaries to ensure that the area can be properly restored to the screen. It then actually uses an assembly-language routine (PutCopyBox) to restore the screen background. When used with MENU.BAS and WINDOW.BAS, buffers created for each menu or window contain the background for the area occupied by the menu or window. When the menu or window is no longer displayed, the background for that area is restored to the screen.

See Also **GetBackground**, **MenuDo**, **WindowDo**, **WindowRefresh**, **WindowShadowRefresh**

Scroll SUB

Action Scrolls the defined area.

Syntax **Scroll** *row1%, col1%, row2%, col2%, lines%, attr%*

Remarks **Scroll** is used in WINDOW.BAS to scroll an area within a defined window. The **Scroll** procedure uses the following arguments:

Argument	Description
row1%, col1%	An integer pair that specifies absolute screen row and column coordinates.
row2%, col2%	An integer pair that specifies the lower-right corner coordinates of an area.

lines% An integer that defines the scrolling behavior that occurs within the defined area. If *lines%* is less than 0, the area scrolls down by that number of lines. If *lines%* is greater than 0, the area scrolls up by that number of lines. If *lines%* is 0, the defined area is cleared.

attr% An integer in the range 0–7 that defines the color to fill any newly created blank lines.

See Also **WindowScroll**

Compatibility with Microsoft QuickBASIC for the Macintosh

Every effort has been made to retain a high degree of compatibility between the functionality in the User Interface toolbox and that which exists in QuickBASIC for the Apple Macintosh. Many programs from the Macintosh platform can be readily converted for use with the PC. However, there are some limitations that you should be aware of.

Programs that use graphics windows on the Macintosh cannot be converted because this user interface package is text (character) based.

Programs that use event trapping on the Macintosh must be converted to polling before conversion can be effected.

Each Macintosh QuickBASIC window command has a functional counterpart in the User Interface toolbox. Often a Macintosh QuickBASIC command has more than one function. Where this is true, there may be more than one corresponding procedure for QuickBASIC on the PC. The following table can be used as a general guideline for converting programs from one platform to the other:

Macintosh QuickBASIC	PC BASIC
ALERT	**Alert**
BUTTON function	**ButtonInquire**
BUTTON statement	**ButtonOpen**, **ButtonSetState**, **ButtonToggle**
BUTTON CLOSE	**ButtonClose**
CLS	**WindowCls**
COMMAND	**ShortCutKeyDelete**, **ShortCutKeyEvent**, **ShortCutKeySet**
DIALOG	**Dialog**, **WindowDo**
EDIT$	**EditFieldInquire**
EDIT FIELD	**EditFieldOpen**
EDIT FIELD	**CLOSE**, **EditFieldClose**
LOCATE	**WindowLocate**
PRINT	**WindowPrint**

MENU function	**MenuCheck, MenuEvent, MenuInkey$**
MENU statement	**MenuColor, MenuItemToggle, MenuPreProcess, MenuSet, MenuSetState, MenuShow**
MENU RESET	**MenuInit**
MENU ON	**MenuOn**
MENU OFF	**MenuOff**
MOUSE	**MouseHide, MouseInit, MousePoll, MouseShow**
SCROLL	**WindowScroll**
WINDOW function	**WindowCols, WindowCurrent, WindowRows**
WINDOW statement	**WindowColor, WindowInit, WindowOpen, WindowSetCurrent**
WINDOW CLOSE	**WindowClose**
Miscellaneous	**WindowBox, WindowLine**

A number of features are similar, but not exactly the same in both environments. Some of the differences are outlined below:

- Buttons cannot be disabled on the PC, only opened or closed.
- On the Macintosh, the active window may differ from the current output window. On the PC, the active window and the current window are the same.
- On the PC, the `Dialog(0)` procedure returns 2 whenever an edit field is selected, not just when there is more than one edit field, as occurs on the Macintosh.
- On the PC, edit fields cannot be centered or right justified. They are always left justified.
- Macintosh **MOUSE** functions 3 through 6 are not implemented on the PC.
- Macintosh **WINDOW** type 7 is not implemented in any manner on the PC. Window types are somewhat different on the PC than they are on the Macintosh. Each of the windows on the Macintosh—numbered 1 through 7—has a slightly different functionality. The PC window types allow you to specify any combination of available features with **WindowOpen**, so that you have full control over the features of your PC windows. Custom window configurations, beyond the 24 basic types provided, can be created by modifying the **WindowBorder** procedure.

A number of new or extended features have been implemented to enhance the usability of this package. These features are described as follows:

- The **Dialog** procedure has been greatly extended to cover the full range of menu and keyboard events.
- All menus are keyboard accessible by two means. Access keys (standard) are provided and shortcut keys (optional) can be user defined.
- Print formatting (word wrap, centering, and so on) can be specified.
- Window contents are automatically saved when windows overlap, so windows do not need to be refreshed. An exception is that a window must be refreshed when a window is resized. The programmer must ensure that refreshing occurs when a window is resized.
- Buttons and menus can be toggled.
- Scroll bars are implemented as button types 6 (vertical) and 7 (horizontal).
- List boxes are implemented.
- Field buttons (invisible buttons that occupy an area of the screen) are implemented as button type 4.

One very important difference between the PC and the Macintosh lies in the use of the keyboard. On the Macintosh, the assumption is made that every computer has a mouse pointing device. This assumption cannot be made for the PC. During the evolution of text window environments on the PC, a standard has emerged that is adhered to by the procedures in this toolbox.

This creates a significant problem when converting programs from the Macintosh to the PC. Keys with well defined purposes, like the Spacebar and the Tab key, actually get trapped by the routines that process keyboard input. This means that Macintosh applications that are particularly keyboard intensive will not work well in the PC Windows environment. A word processor, for example, would be abysmally slow and therefore would not be practical within a window. There are three choices available for the programmer who wants to convert such an application:

- Do all the keyboard-intensive operations outside of the **WindowDo** procedure. The vast majority of PC window-based applications do this anyway. Then you can use windows for dialog and list boxes. This is the preferred method.
- Learn how **WindowDo** works and customize it, or write your own **WindowDo** procedure to suit your needs.
- Learn how edit fields and buttons operate, and create your own object type. The edit field can be extended to support multiple lines.

Generally, these routines are quite flexible and can certainly add functionality to your programs, as well as conversion capability from one platform to the other.

Appendixes

Appendix A
Keyboard Scan Codes and ASCII Character Codes

Keyboard Scan Codes

The table on the next two pages shows the DOS keyboard scan codes. These codes are returned by the **INKEY$** function.

Key combinations with NUL in the Char column return two bytes — a null byte (&H00) followed by the value listed in the Dec and Hex columns. For example, pressing Alt+F1 returns a null byte followed by a byte that contains 104 (&H68).

Following the Keyboard Scan Codes table is a table that lists ASCII character codes.

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B		27	1B		27	1B				
1 !	2	02	49	31	1	33	21	!				120	78	NUL
2 @	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3 #	4	04	51	33	3	35	23	#				122	7A	NUL
4 $	5	05	52	34	4	36	24	$				123	7B	NUL
5 %	6	06	53	35	5	37	25	%				124	7C	NUL
6 ^	7	07	54	36	6	94	5E	^	30	1E		125	7D	NUL
7 &	8	08	55	37	7	38	26	&				126	7E	NUL
8 *	9	09	56	38	8	42	2A	*				127	7F	NUL
9 (	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
- _	12	0C	45	2D	-	95	5F	_	31	1F		130	82	NUL
= +	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F				
TAB	15	0F	9	09		15	0F	NUL						
Q	16	10	113	71	q	81	51	Q	17	11		16	10	NUL
W	17	11	119	77	w	87	57	W	23	17		17	11	NUL
E	18	12	101	65	e	69	45	E	5	05		18	12	NUL
R	19	13	114	72	r	82	52	R	18	12		19	13	NUL
T	20	14	116	74	t	84	54	T	20	14		20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19		21	15	NUL
U	22	16	117	75	u	85	55	U	21	15		22	16	NUL
I	23	17	105	69	i	73	49	I	9	09		23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F		24	18	NUL
P	25	19	112	70	p	80	50	P	16	10		25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B				
] }	27	1B	93	5D	]	125	7D	}	29	1D				
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF			
CTRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01		30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13		31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04		32	20	NUL
F	33	21	102	66	f	70	46	F	6	06		33	21	NUL
G	34	22	103	67	g	71	47	G	7	07		34	22	NUL
H	35	23	104	68	h	72	48	H	8	08		35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A		36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B		37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C		38	26	NUL
; :	39	27	59	3B	;	58	3A	:						
' "	40	28	39	27	'	34	22	"						
` ~	41	29	96	60	`	126	7E	~						

† Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
L SHIFT	42	2A												
\\|	43	2B	92	5C	\	124	7C	|	28	1C				
Z	44	2C	122	7A	z	90	5A	Z	26	1A		44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18		45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03		46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16		47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02		48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E		49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D		50	32	NUL
, <	51	33	44	2C	,	60	3C	<						
. >	52	34	46	2E	.	62	3E	>						
/ ?	53	35	47	2F	/	63	3F	?						
R SHIFT	54	36												
* PRTSC	55	37	42	2A	*		INT 5§		16	10				
ALT	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	46	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
UP	72	48	72	48	NUL	56	38	8						
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
GREY -	74	4A	45	2D	-	45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
GREY +	78	4E	43	2B	+	43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
DOWN	80	50	80	50	NUL	50	32	2						
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
INS	82	52	82	52	NUL	48	30	0						
DEL	83	53	83	53	NUL	46	2E	.						

† Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

§ Under DOS, SHIFT + PRTSC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

ASCII Character Codes

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	32	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL-BKSP key.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	232	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Appendix B
BASIC Reserved Words

The words in the following list are reserved in BASIC. They may not be used as labels or as names for variables or procedures. Reserved words include keywords (functions, statements and operators) and certain other words that are recognized by the interpreter.

ABS
ACCESS
ALIAS
ALL
AND
ANY
APPEND
AS
ASC
ATN

BASE
BEEP
BEGINTRANS
BINARY
BLOAD
BOF
BSAVE
BYVAL

CALL
CALLS
CASE
CCUR
CDBL
CDECL
CHAIN
CHDIR
CHDRIVE
CHECKPOINT
CHR$
CINT
CIRCLE
CLEAR
CLNG
CLOSE
CLS
COLOR
COM
COMMAND$
COMMITTRANS
COMMON
CONST
COS
CREATEINDEX
CSNG
CSRLIN
CURDIR$
CURRENCY
CVC
CVD
CVDMBF
CVI
CVL
CVS
CVSMBF

DATA
DATE$
DECLARE
DEF
DEFCUR
DEFDBL
DEFINT
DEFLNG
DEFSNG
DEFSTR
DELETE
DELETEINDEX
DELETETABLE
DIM
DIR$
DO
DOUBLE
DRAW

ELSE
ELSEIF
END
ENDIF
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
EVENT
EXIT
EXP

FIELD
FILEATTR
FILES
FIX
FOR
FRE
FREEFILE
FUNCTION

GET
GETINDEX$
GO
GOSUB
GOTO

HEX$

IF
IMP
INKEY$
INP
INPUT
INPUT$
INSERT
INSTR
INT
INTEGER
IOCTL
IOCTL$
IS
ISAM

KEY
KILL

LBOUND
LCASE$
LEFT$
LEN
LET
LINE
LIST
LOC
LOCAL
LOCATE
LOCK

LOF
LOG
LONG
LOOP
LPOS
LPRINT
LSET
LTRIM$

MID$
MKC$
MKD$
MKDIR
MKDMBF$
MKI$
MKL$
MKS$
MKSMBF$
MOD
MOVEFIRST
MOVELAST
MOVENEXT
MOVEPREVIOUS

NAME
NEXT
NOT

OCT$
OFF
ON
OPEN
OPTION
OR
OUT
OUTPUT

PAINT
PALETTE
PCOPY
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PSET
PUT

RANDOM
RANDOMIZE
READ
REDIM
REM
RESET
RESTORE
RESUME
RETRIEVE
RETURN
RIGHT$
RMDIR
RND
ROLLBACK
RSET
RTRIM$
RUN

SADD
SAVEPOINT
SCREEN
SEEK
SEEKEQ
SEEKGE
SEEKGT
SEG
SELECT
SETINDEX
SETMEM
SGN
SHARED
SHELL
SIGNAL
SIN
SINGLE
SLEEP
SOUND
SPACE$
SPC
SQR
SSEG
SSEGADD
STACK
STATIC
STEP
STICK
STOP
STR$
STRIG
STRING
STRING$
SUB
SWAP
SYSTEM

TAB
TAN
THEN
TIME$
TIMER
TO
TROFF
TRON
TYPE

UBOUND
UCASE$
UEVENT
UNLOCK
UNTIL
UPDATE
USING

VAL
VARPTR
VARPTR$
VARSEG
VIEW

WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE

XOR

Appendix C
Command-Line Tools Quick Reference

This appendix lists the syntax and options for the following Microsoft BASIC command-line tools:

■ BC	The Microsoft BASIC Compiler.
■ BUILDRTM	The utility that creates custom run-time modules.
■ HELPMAKE	The utility that creates or modifies help files.
■ LIB	The utility that creates run-time libraries.
■ LINK	The utility that links object files and libraries.
■ MKKEY	The utility that converts a .KEY file into an ASCII file or and ASCII file into a .KEY file.
■ NMAKE	The utility that automates compiling and linking.
■ QBX	The integrated development environment.

Valid options for these tools are listed alphabetically together with a brief description of their function. For detailed descriptions of specific command options, see the *Programmer's Guide*.

BASIC Compiler (BC)

Action Compiles BASIC source code.

Syntax BC *sourcefile* [[*objectfile*]] [[*listingfile*]] [[*optionlist*]] [[;]]

Remarks The Microsoft BASIC Compiler (BC) uses the following arguments:

Argument	Description
sourcefile	The name of your BASIC source code file. If USER is specified, input is taken from the keyboard.
objectfile	The name of the object file you are creating.

listingfile	The name of the file that will contain an assembly-code source listing of the compiler-generated code. The file contains the address of each line in your source file, the text of the source file, its size, and any error messages produced during compilation. If USER is specified, the listing is sent to the screen.
optionlist	One or more of the BC command-line options outlined in the table on the following page.

The BC command-line options are listed in the following table:

Option	Description
/A	Creates a listing of the disassembled object code for each source line and shows the assembly-language code generated by the compiler.
/Ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to occupy be larger than 64K each. If this option is not specified, the maximum size is 64K per dynamic array. You must use either /Ah or /D when you are compiling Quick Library routines that will be loaded into QBX with the /Ea option (which moves arrays into expanded memory).
/C:*buffersize*	Sets the size of the buffer for each communications port receiving remote data when using an asynchronous communications adapter.
/D	Generates debugging code for run-time error checking; enables Ctrl+Break. For ISAM programs, causes BASIC to write open database buffers to disk after every **DELETE**, **INSERT**, **UPDATE** and **CLOSE** statement. You must use either /Ah or /D when you are compiling Quick Library routines that will be loaded into QBX with the /Ea option (which moves arrays into expanded memory).
/E	Specifies presence of **ON ERROR** with **RESUME** *linenumber* statements. (See also the discussion of the /X option below.)
/Es	Allows you to share expanded memory between BASIC and mixed-language routines that make use of expanded memory. Specify /Es when you are going to use a mixed-language routine that makes use of expanded memory.
/FPa	Causes your program to use the alternate-math library for floating-point operations.
/FPi	Causes the compiler to generate "in-line instructions" for use in floating-point operations (default).
/Fs	Enables far strings in user programs.
/G2	Generates 80286-specific instructions.

/Ib	The Ib:, /Ie:, and /Ii: options control memory management for ISAM. For more information, see Chapter 10, "Database Programming with ISAM" in the *Programmer's Guide*. The /Ib option sets the minimum number of buffers to be used by ISAM. ISAM defaults to a minimum of 6 buffers for reduced functionality (PROIASM) and 9 for full functionality (PROIASMD). The maximum allowable value is 512.
/Ie	Sets the amount of expanded memory, in kilobytes, to leave for non-ISAM use when you are working with ISAM. If omitted, all expanded memory (up to about 1.2 megabytes) is used by ISAM. A value of -1 indicates ISAM should use no expanded memory.
/Ii	Specifies the maximum number of non-null ISAM indexes in tables used in a program. If this argument is omitted, the default value is 30, which is also the minimum. The maximum value is 500.
/Lp	Creates a protected-mode object file (the default if running BC from an OS/2 protected-mode session).
/Lr	Creates a real-mode object file (the default if running BC from DOS or an OS/2 real-mode session).
/MBF	Converts the intrinsic functions **MKS$**, **MKD$**, **CVS**, and **CVD** to **MKSMBF$**, **MKDMBF$**, **CVSMBF**, and **CVDMBF**, respectively. This allows your BASIC program to read and write floating-point values stored in Microsoft Binary format.
/O	Substitutes the default stand-alone library for the default run-time library (creates a stand-alone executable file that can run without a BASIC run-time module).
/Ot	Optimizes execution speed for **SUB** and **FUNCTION** procedures and **DEF FN** statements. To use this type of optimizing, certain conditions must be met. The frame size generated for **SUB** and **FUNCTION** procedures, and statements defined by **DEF FN**, depend on which of the following conditions exist in your code. For **SUB** and **FUNCTION** procedures: A reduced frame is generated with /Ot if no module-level error handler exists in the code, and the /D or /Fs option isn't used. A full frame is generated if your code uses local error handlers, calls a **DEF FN** or **GOSUB** statement, has a return (because of a **GOSUB** or other reason), or contains an **ON** *event* **GOSUB**. For statements defined by **DEF FN**: A full frame is generated if the /D, /Fs, /E, or /X option is used. A partial frame is generated if the /W or /V option is used. In all other cases, no frame is generated.

/R	Stores arrays in row-major order. BASIC normally stores arrays in column-major order.
/S	Writes quoted strings directly to the object file instead of the symbol table. Use this option when an `Out of memory` error message is generated while BC is compiling a program that has many string constants.
/T	Suppresses warnings given by the compiler. By default the interpreter passes this option to BC when you select Make EXE file in the QBX environment.
/V	Enables event trapping for communications (**COM**), lightpen (**PEN**), joystick (**STRIG**), timer (**TIMER**), music-buffer (**PLAY**), function-key (**KEY**), OS/2 signal events (**SIGNAL**), and user-defined (**UEVENT**) events. Checks between each BASIC statement for events.
/W	Enables event trapping for the same statements as /V, but checks at each line number or label for the occurrence of an event.
/X	Specifies presence of **ON ERROR** with **RESUME**, **RESUME NEXT**, or **RESUME 0**.
/Z	Produces a listing of compile-time errors in a form readable by the M editor. If used when compiling within M, the /Z option allows you to locate and scroll through errors in your source and include files by invoking the *nxtmsg* editor function.
/Zd	Produces an object file that contains line-number records for debugging purposes corresponding to the line numbers of the source file.
/Zi	Adds debugging information to the object file that can be used by the Microsoft CodeView® debugger.

Note If you are compiling your program from within the QBX environment, not all BC options can be specified. See the Run menu's Make Exe File dialog box for options that can be specified, and their current settings.

BUILDRTM Utility

Action Creates a custom run-time module.

Syntax BUILDRTM [[*options*]] {/DEFAULT | *runtime exportlist*}

Remarks The **BUILDRTM** utility uses the following arguments:

Argument	Description
options	A list of command options to BUILDRTM, separated by spaces.
/DEFAULT	Specifies the building of a default run-time module.
runtime	A string that contains the filename—without the extension—for the custom run-time module.
exportlist	A string that contains the filename of your export list. See Chapter 21, "Building Custom Run-Time Modules" in the *Programmer's Guide* for the proper syntax for *exportlist*.

Possible *options* values for BUILDTRM are:

Option	Description
/H [[ELP]]	Displays help about BUILDRTM options and required files.
/FP*method*	Specifies the floating-point-math method used. The argument *method* can be /FPa (alternate math), /FPi (in-line instructions), or /FPi87(math-coprocessor library). The default is /FPi.
/L*mode*	Specifies the target environment. Possible modes are /Lr (real mode) and /Lp (protected mode). The current mode is the default.
/Fs	Enables far-string support.
/MAP	Generates a full map file for the custom run-time module.

HELPMAKE Utility

Action Creates or modifies help files for use with BASIC or other Microsoft language products.

Syntax HELPMAKE [[*options*]] { /E [[*n*]] | /D [[*letter*]] } {*sourcefiles*}

Remarks The HELPMAKE utility uses the following arguments:

Argument	Description
options	A list of command options to HELPMAKE.
/E [[*n*]]	Creates (encodes) a help database from a specified text file. The argument *n* indicates the type of compression to take place, and can be a combination of techniques, where *n* is the summation of the individual values listed below:

Value	Compression technique
0	No compression
1	Run-length compression
2	Keyword compression
4	Extended-keyword compression
8	Huffman compression

Argument	Description
/D [[*letter*]]	Decodes the input file into its component parts. Possible values for *letter* are:

Value	Decode effect
(None specified)	Fully decodes help database, leaving intact all cross-references and formatting information.
S	(Decode Split) Splits the concatenated, compressed help database into its components, using their original names.
U	(Decode Unformatted) Decompresses the database and removes all screen formatting and cross-references.

Argument	Description
sourcefiles	A list of the input file(s) to be encoded or decoded.

Possible *options* values for encoding/decoding a help database are as follows:

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the help-database file.
/C	Indicates that the context strings for this help file are case sensitive.
/H	Displays a summary of HELPMAKE syntax and exits without encoding any files.
/L	Locks the generated file so that it cannot be decoded by HELPMAKE at a later time.
/O*destfile*	Specifies *destfile* as the name of the help database (encoding) or the destination file for the decoded output from HELPMAKE (decoding).
/S*n*	Specifies the type of input file. Possible values for *n* are:

Value	File Type
1	Rich Text Format (RTF)
2	QuickHelp (the default)
3	Minimally formatted ASCII

Option	Action
/T	Enables translation of a number of dot (.) commands. Each of these commands corresponds to a statement using the application control character, which by default is a colon (:). For example, .length 20 is equivalent to :120. Use of the /T option directs HELPMAKE to perform these translations, as described in the following table:

Dot command	Translated to:
.category	:c
.command	:x
.dup	:d
.end	:e
.execute	:y
.file	:f
.freeze	:z
.length	:l
.list	:i
.next	:>
.mark	:m
.paste	:p
.popup	:g
.previous	:<
.ref	:r
.suffix	:s
.topic	:n

/V ⟦*n*⟧	Specifies the amount of diagnostic and informational output to be displayed. Possible values for *n* are:

Value	Effect
(*n* omitted)	Maximum diagnostic output
0	No diagnostic output and no banner
1	Prints only HELPMAKE banner (default)
2	Prints pass names
3	Prints contexts on first pass
4	Prints contexts on each pass
5	Prints any intermediate steps within each pass
6	Prints statistics on help file and compression

/W*width*	Specifies the number of characters for the fixed width of the resulting help text. The value of *width* may range from 11 to 255, inclusive.

LIB Utility

Action Creates, organizes, and maintains run-time libraries.

Syntax LIB *oldlibrary*⟦*options*⟧ ⟦*commands*⟧ ⟦, ⟦*listfile*⟧ ⟦, ⟦*newlibrary*⟧⟧⟧ ⟦;⟧

Remarks The LIB utility uses the following arguments:

Argument	Description
oldlibrary	The name of the existing library that will be operated upon.
options	A list of command options to LIB.
commands	The command symbols used for manipulating modules.
listfile	The name of the cross-reference-listing file to be generated.
newlibrary	The name of the modified library to be created by LIB.

Possible values for *options* are as follows:

Option	Description
/PA [[GESIZE]]: *number*	Specifies the library-page size of a new library, or changes the library-page size of an existing library. The default page size for a new library is 16 bytes.
/NOI [[GNORECASE]]	Directs LIB not to ignore case when comparing symbols.
/I [[GNORECASE]]	Directs LIB to ignore case when comparing symbols (the default). Use to combine a library marked /NOI with an unmarked library to create a new unmarked library.
/HELP	Causes LIB to attempt to execute QH.EXE. If unsuccessful, a usage message will be displayed.
/NOE [[XTDICTIONARY]]	Prevents LIB from creating an extended dictionary.
/NOLOGO	Causes LIB to suppress the sign-on banner.

The following table describes the LIB command codes (possible values for *commands*):

Code	Task	Description
+	Add	Appends an object file or library file to the given library.
–	Delete	Deletes a module from the library.
–+	Replace	Replaces a module by deleting it from the library and appending to the library an object file with the same name.
*	Copy	Extracts a module without deleting it from the library and saves the module as an object file with the same name.
–*	Move	Extracts a module and deletes it from the library after saving it in an object file with the same name.

LINK Utility

Action Combines object files and libraries into a single executable file or a Quick library.

Syntax LINK ⟦*options*⟧ *objfiles* ⟦, ⟦*exefile*⟧ ⟦, ⟦*mapfile*⟧ ⟦, ⟦*libraries*⟧ ⟦, ⟦*deffile*⟧⟧⟧⟧⟧ ⟦;⟧

Remarks The LINK utility uses the following arguments:

Argument	Description
options	A list of command options to LINK. They may appear anywhere on the command line following the LINK command.
objfiles	The object modules (.OBJ) or libraries (.LIB) to be linked. Multiple object files are separated with spaces or plus signs (+).
exefile	The name of the executable file to be created.
mapfile	The map file to be created.
libraries	The libraries that you want linked with the object file.
deffile	The module-definition file (for OS/2 protected mode).

If a semicolon (;) appears after a field, LINK uses default values for the remaining fields; if no semicolon appears, LINK prompts the user for the remaining parameters.

The following LINK options can be used with BASIC programs:

Option	Description
/A ⟦LIGNMENT⟧:*size*	Aligns segment data according to *size*.
/BA ⟦TCH⟧	Prevents LINK from prompting for path when it cannot find specified libraries or object files.
/CO ⟦DEVIEW⟧	Prepares program for debugging with Microsoft CodeView.
/DO ⟦SSEG⟧	Forces a special ordering on segments. Not for use in custom run-time modules.
/E ⟦XEPACK⟧	Packs executable files during linking.
/F ⟦ARCALLTRANSLATION⟧	Optimizes far calls to procedures in the same segment as the caller. Use in conjunction with /PACKCODE. This option is recommended for reducing the size of an executable file.
/HE ⟦LP⟧	Lists LINK options to standard output.
/INF ⟦ORMATION⟧	Displays linker-process information.

/LI ⟦NENUMBERS⟧	Includes line numbers in the map file.
/M ⟦AP⟧	Lists public symbols defined in the object file.
/NOD ⟦EFAULTLIBRARYSEARCH⟧	Ignores default libraries during linking.
/NOE ⟦XTDICTIONARY⟧	Ignores extended dictionary during linking.
/NOF ⟦ARCALLTRANSLATION⟧	Disables far-call optimizing (/FARCALLTRANSLATION).
/NOI ⟦GNORECASE⟧	Preserves case sensitivity. Not for use in OS/2 custom run-time modules.
/NOL ⟦OGO⟧	Suppresses the sign-on logo.
/NON ⟦ULLDOSSEG⟧	Works the same as the /DOSSEG option, except that no null bytes are inserted at the beginning of the _TEXT segment.
/NOP ⟦ACKCODE⟧	Disables segment packing (/PACKCODE).
/O ⟦VERLAYINTERRUPT⟧:*number*	Specifies an interrupt *number* other than 0x3F for passing control to overlays.
/PACKC ⟦ODE⟧	Packs contiguous code segments. Not for use with overlays.
/PACKD ⟦ATA⟧	Packs contiguous data segments.
/PAU ⟦SE⟧	Causes LINK to pause before writing the executable file to disk.
/PMTYPE:*type*	Specifies OS/2 program type, where *type* can be the following: PM (not valid for BASIC programs), VIO (valid with restrictions), or NOVIO (default, valid for BASIC). See the full description in Chapter 18, "Using LINK and LIB" in the *Programmer's Guide* for further information and restrictions.
/Q ⟦UICKLIBRARY⟧	Produces a Quick library for use with QBX.
/SE ⟦GMENTS⟧:*number*	Sets maximum number of segments that the linker allows a program to have.
/W ⟦ARNFIXUP⟧	Issues fixup warnings. Not for use with custom run-time modules. For BASIC programs, must be used in conjunction with /NOPACKCODE.

Not all options of the LINK command are suitable for use with BASIC programs. Following are options that do not have an effect or that should not be used with BASIC programs:

Option	Action
/CP [[ARMAXALLOC]]:*number*	Sets the maximum *number* of 16-byte paragraphs needed by the program when it is loaded into memory. Although you can use this option, it has no effect, because while it is running, your BASIC program controls memory.
/DS [[ALLOCATE]]	Loads all data starting at the high end of the default data segment.
/HI [[GH]]	Places the executable file as high in memory as possible.
/ST [[ACK]]:*number*	Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. The standard BASIC library sets the default stack size to 3K for DOS and 3.5K for OS/2. BASIC programs should use the **STACK** or **CLEAR** statement instead.
/T [[INY]]	Produces .COM files for small programs where DOS load time is an issue.

Note

The /DS and /HI options are suitable only for object files created by the Microsoft Macro Assembler (MASM).

MKKEY Utility

Action Converts a .KEY file into an ASCII file that you can edit or converts an ASCII file into a .KEY file.

Syntax MKKEY -c {*ab* | *ba*} -i *inputfilename* -o *outputfilename*

Remarks The MKKEY utility uses the following arguments:

Argument	Description
-c	Specifies the type of conversion, either ASCII to binary or binary to ASCII.
-i	Specifies the input file.
-o	Specifies the output file.

If you prefer to use a set of editing commands other than the defaults, Microsoft BASIC comes with four predefined key files and a utility for making your own key file.

The four key files are QBX.KEY, ME.KEY, BRIEF.KEY, and EPSILON.KEY. These files contain slightly different key assignments (also known as key bindings). For example, the QBX editor uses Ctrl+E to move the cursor up, while Epsilon uses Ctrl+P. The definitions in QBX.KEY are the same as the default definitions you get without using any .KEY file.

A complete list of the key bindings for the four .KEY files can be found online under the Contents, Configuring Keys help topic. A complete description of the action of each function also can be found there.

To use anything other than the default key definitions you must specify a .KEY file. Use the /k: option when you start QBX to load the .KEY file that contains the desired key definitions. For example, type the following to load the BRIEF.KEY file:

```
QBX /k:BRIEF.KEY
```

Your preference is saved in the QBX.INI file. You don't need to specify the key file in future editing sessions.

To modify the QBX.KEY file, you first convert it to an editable ASCII file by specifying binary to ASCII conversion:

```
MKKEY -c ba -i QBX.KEY -o MYEDITOR.TXT
```

You can use any text editor (including the QBX editor) to edit the file named MYEDITOR.TXT, which lists the keystrokes that perform certain actions. For example, QBX.KEY assigns or binds Ctrl+G to Del, a function that deletes a single character. This is done with the line:

```
Del : CTRL+G
```

You can change that command to any other key (for example, Ctrl+D) as long as that key isn't already assigned (bound) to another function. If Ctrl+D is assigned to CharRight elsewhere in the file (as it is in the original QBX.KEY file), delete or change that line if you really want to use Ctrl+D for the Del function. You can have several keys perform the same function, but you cannot have the same key perform more than one function.

When you're satisfied with the new functions, convert the ASCII file to binary so it can be loaded into the QBX editor:

```
MKKEY -c ab -i MYEDITOR.TXT -o MYEDITOR.KEY
```

Finally, use the /k: option described above to load the new key file.

NMAKE Utility

Action Automates the process of compiling and linking project files.

Syntax NMAKE ⟦*options*⟧ ⟦*macrodefinitions*⟧ ⟦*target...*⟧ ⟦*filename*⟧

Remarks The NMAKE utility uses the following arguments:

Argument	Description
options	A list of command options to NMAKE.
macrodefinitions	The name of one or more targets to build. If you do not list any targets, NMAKE builds the first target in the description file.
target...	The name of one or more targets to build. If you do not list any targets, NMAKE builds the first target in the description file.

filename	An optional field that gives the name of the description file from which NMAKE reads target- and dependent-file specifications and commands. By default, NMAKE looks for a file named MAKEFILE in the current directory. If MAKEFILE does not exist, NMAKE uses the *filename* field in the following manner. It interprets the first string on the command line that is not an option or macro definition as the name of the description file (provided its filename extension isn't listed in the .SUFFIXES list).

The NMAKE options are described in the following table:

Option	Action
/A	Executes commands to build all the targets requested even if they are not out of date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from being displayed.
/D	Displays the modification date of each file when the date is checked.
/E	Causes environment variables to override macro definitions within description files.
/F *filename*	Specifies *filename* as the name of the description file to use. If a dash (–) is entered instead of a filename, NMAKE accepts input from the standard input device instead of using a description file. If /F is not specified, NMAKE uses MAKEFILE as the description file. If MAKEFILE does not exist, NMAKE uses as the filename the first string on the command line that is not an option or macro definition and that does not have an extension in the .SUFFIXES list.
/HELP	Displays help information about NMAKE syntax and options.
/I	Ignores exit codes (also called return or error codes) returned by programs called from the NMAKE description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays the commands from the description file that NMAKE would execute, but does not execute these commands. This option is useful for checking which targets are out of date, and for debugging description files.
/NOLOGO	Suppresses the NMAKE copyright message.
/P	Prints all macro definitions and target descriptions.

/Q	Returns a zero status code if the target is up to date, and a non-zero status code if it is not. This option is useful when invoking NMAKE from within a batch file.
/R	Ignores inference rules and macros contained in the TOOLS.INI file.
/S	Suppresses display of commands as they are executed.
/T	Changes the modification dates for out-of-date target files to the current date. The file contents are not modified.
/X *filename*	Sends all error output to *filename*, which can be either a file or a device. If a dash (–) is entered instead of a filename, the error output is sent to the standard output device.

Description Blocks

NMAKE description blocks use the following format:

target...:⟦*dependent...*⟧ ⟦;*command*⟧ ⟦#*comment*⟧
⟦*command*⟧
⟦#*comment*⟧
⟦{#*comment* | *command*}⟧
.
.
.

The following symbols are used in description blocks:

Symbol	Meaning
#	Introduces a comment field.
* ?	The DOS wild-card characters; NMAKE expands them in target names when it reads the description file.
^ (Caret)	Introduces any escape character in a description file, including these: # ( ) $ ^\ ! @

Command Modifiers

These characters can be placed in front of a command to modify its effect. The character must be preceded by at least one space.

Character	Action
– (Dash)	Turns off error checking for the command.
@ (At sign)	Prevents NMAKE from displaying the command as it executes.
! (Exclamation point)	Causes the command to be executed for each dependent file if the command uses one of the special macros $? or $**.

Macro Definitions

A macro definition uses the following syntax:

macroname=string

The argument *macroname* can be any combination of alphanumeric characters and the underscore (_) character. The argument *string* can be any valid string.

Having defined the macro, use the following reference to include it in a dependency line or command:

$(*macroname*)

Use the following syntax to substitute text within a macro:

$(*macroname:string1 = string2*)

Special Macro Names

The following macro names have special meanings:

Macro	Meaning
$*	The target name with the extension deleted.
$@	The full name of the current target.
$**	The complete list of dependent files.
$<	The dependent file that is out of date with respect to the target (evaluated only for inference rules).
$?	The list of dependents that are out of date with respect to the target.
$$@	The target NMAKE is currently evaluating. A dynamic dependency parameter used only in dependency lines.
$(CC)	The command to invoke the C compiler. By default, NMAKE predefines this macro as `CC = CL.`
$(AS)	The command that invokes the Microsoft Macro Assembler. NMAKE predefines this macro as `AS = masm.`
$(BC)	The command to invoke the Microsoft BASIC compiler, BC. NMAKE predefines this macro as `BC = bc.`

$(MAKE)	The name with which NMAKE is invoked. Used to invoke NMAKE recursively; It causes the line on which it appears to be executed even if the /N option is on. Redefine this macro if you want to execute another program.
$(MAKEFLAGS)	The NMAKE options currently in effect. If you invoke NMAKE recursively, you should use $(MAKE). You cannot redefine this macro.

Inference Rules

Inference rules are templates that NMAKE uses to generate files with a given extension. They have the following syntax:

fromext.toext:
command
⟦*command*⟧
.
.
.

NMAKE uses these predefined inference rules:

Inference rule	**Command**	**Default action**
.bas.obj	$(BC) $(BFLAGS)$*.bas;	BC $*.bas;
.c.obj	$(CC) $(CFLAGS) /c $*.c	CL /c $*.c
.c.exe	$(CC) $(CFLAGS) $*.c	CL $*.c
.asm.obj	$(AS) $(AFLAGS) $*;	masm $*;

Directives

The following directives conditionally execute commands, display error messages, include the contents of other files, and turn on or off some of NMAKE's options:

Directive	**Description**
!IF *expression*	Executes the statements between the !IF keyword and the next !ELSE or !ENDIF directive if *expression* evaluates to a non-zero value. The argument *expression* consists of integer constants, string constants, or program invocations. Integer constants can use the C unary operators for numerical negation (–), one's complement (~), and logical negation (!), and the C binary operators.

!ELSE	Executes the statements between the !ELSE and !ENDIF directives if the statements preceding the !ELSE directive were not executed.
!ENDIF	Marks the end of the !IF, !IFDEF, or !IFNDEF block of statements.
!IFDEF *macroname*	Executes the statements between the !IFDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is defined in the description file. NMAKE considers a macro with a null value to be defined.
!IFNDEF *macroname*	Executes the statements between the !IFNDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is not defined in the description file.
!UNDEF *macroname*	Marks *macroname* as being undefined in NMAKE's symbol table.
!ERROR *text*	Causes *text* to be printed and then stops execution.
!INCLUDE *filename*	Reads and evaluates the file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (< >), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise it looks in the current directory only. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.
!CMDSWITCHES: + | – [[*options*]]	Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If no options are specified, the options are reset to the way they were when NMAKE was started. Turn an option on by preceding it with a plus sign (+), or turn it off by preceding it with a minus sign (–). Using this directive updates the MAKEFLAGS macro.

Pseudotargets

A "pseudotarget" is a name that serves as a handle for building a group of files or executing a group of commands. The NMAKE utility includes these four predefined pseudotargets that provide special rules within a description file.

Pseudotarget	Action
.SILENT:	Does not display lines as they are executed. Has the same effect as invoking NMAKE with the /S option.
.IGNORE:	Ignores exit codes returned by programs called from the description file. Has the same effect as invoking NMAKE with the /I option.
.SUFFIXES:*list*	Lists file suffixes for NMAKE to try if it needs to build a target file for which no dependents are specified.
.PRECIOUS: *target*...	Tells NMAKE not to delete *target* if the commands that build it are interrupted. Overrides the NMAKE default.

QuickBASIC Extended (QBX)

Action Invokes the integrated development environment with a specified program name.

Syntax QBX [[/AH]] [[/B]] [[/C:*buffersize* [[{/Ea | /Es}]] [[/E:*n*]] [[/G]] [[/H]] [[/K:[[*keyfile*]]]]
[[/L [[*libraryname*]]]] [[/MBF]] [[/Nofrills]] [[/NOHI]] [[[[/RUN]] *programname*]] [[/CMD *string*]]

Remarks The following options can be typed on the operating system command line following the QBX command:

Argument	Description
/AH	Allows dynamic arrays of records, fixed-length strings, and numeric data to be larger than 64K each.
/B	Allows the use of a composite (black-and-white) monitor with a color graphics card. The /B option displays QBX in black and white if you have a color monitor.
/C: *buffersize*	Sets the size of the buffer receiving data. This option works only with an asynchronous communications card. The default buffer size is 512 bytes; the maximum size is 32,767 bytes.
/Ea	Allows arrays of any numeric, fixed-length string, or record (user-defined) type to be moved into expanded memory. The array must be ≥ 512 bytes and ≤ 16K to be moved to expanded memory. If /Ea is not specified, no arrays are moved into expanded memory. Do not use /Ea if you pass arrays as parameters to mixed-language Quick library routines. Do not use /Ea with the /Es option. When you invoke QBX with /Ea and /L, the specified Quick library must have been compiled with the BC /Ah or /D option. This is necessary because /Ah and /D cause run-time calls instead of inline code generation for array references.
/E:*n*	Specifies the amount of expanded memory reserved for QBX use, where *n* is the amount of memory in kilobytes. If /E:*n* is not specified, up to all available expanded memory may be used. If /E:0 is used, no expanded memory is used (and the /Ea switch is overridden).
/Es	Allows you to share expanded memory between QBX Quick library and mixed-language routines that make use of expanded memory. Specify /Es when you are going to use a Quick library mixed-language routine that makes use of expanded memory.

/G	Sets QBX to update a CGA screen as fast as possible. The option works only with machines using CGA monitors. If you see snow (dots flickering on the screen) when QBX updates your screen, then your hardware cannot fully support the /G option. If you prefer a clean screen, restart QBX without this option.
/H	Displays the highest resolution possible on your hardware.
/K:[[*keyfile*]]	Specifies a user-configurable key file to be loaded into QBX. A key file has a .KEY filename extension, and defines key mappings that will be used while editing a file in QBX.
/L [[*libraryname*]]	Loads the Quick library that is specified by *libraryname*. If *libraryname* is not specified, the default Quick library, QBX.QLB, is loaded.
/MBF	Causes the QBX conversion functions to treat IEEE-format numbers as Microsoft Binary format numbers.
/Nofrills	The /nofrills option (abbreviated as /nof) lets you make additional memory available for program use. However, it reduces the functionality of the environment. When you start QBX and use the /nof command-line option, QBX uses about 19K less memory but no longer supports the following features: ■ Utility menu and commands. ■ Options menu and commands. ■ Help menu and commands. All of these commands fall into the category of programmer convenience. None is crucial for software development. You can still create any program you could before. The /nof option does not change the structure of the menu bar or status line. However, the three menus listed above are greyed and cannot be opened.
/NOHI	Allows the use of a monitor that does not support high intensity. Not for use with Compaq laptop computers.
programname	Names the file to be loaded when QBX starts.
[[/RUN]] *programname*	Causes QBX to load and run the program before displaying it.
/CMD *string*	Passes string to the COMMAND$ function. This option must be the last option on the line.

Appendix D
Error Messages

During development of a BASIC program, the following types of errors can occur:

- Invocation errors
- Compile-time errors
- Run-time errors
- Link-time errors

Each type of error is associated with a particular step in the program development process:

- Invocation errors occur when you invoke BASIC with the QBX or BC command.
- Compile-time errors (and warnings) occur during compilation. Errors indicate more serious problems than warnings. You can run .EXE files which have only warnings associated with them, whereas an .EXE file (if one is created) associated with an error may or may not run.
- Run-time errors occur when the program is executing.
- Link-time errors occur only when you use the **LINK** command to link object files created with BC or other language compilers.

"Invocation, Compile-Time, and Run-Time Error Messages" lists all the messages alphabetically. Some run-time errors set unique **ERR** values when they occur; these **ERR** values (or "error codes") are included with the run-time error description. Table 4.1 lists the run-time error messages and error codes in numerical order.

"LINK Error Messages" lists the Microsoft Overlay Linker error messages, and "LIB Error Messages" lists the Microsoft Library Manager error messages.

"HIMEM.SYS Error Messages," "RAMDRIVE.SYS Error Messages," and "SMARTDRV.SYS Error Messages" list the error messages for those utilities.

Error-Message Display

When a run-time error occurs within the QBX environment (with default screen options), the error message appears in a dialog box and the cursor is placed on the line where the error occurred.

When a run-time error occurs in a program executed outside of the QBX environment, the error messages appear followed by an address. If any of the /D, /E, or /W options were specified on the BC command line when the program was compiled, then any run-time error messages are followed by the number of the line in which the error occurred. The standard form of this type of error message can be either of the following:

`Error` *n* `in module` *module-name* `at address` *segment:offset*
`Error` *n* `in line` *linenumber* `of module` *module-name* `at address` *segment:offset*

ERR Values

An **ERR** code is listed for some run-time errors. If that error occurs, the value of **ERR** is set to the appropriate code. An error-handling routine can use the value of **ERR** to determine the cause of the error. (Error-handling routines are enabled via the **ON ERROR** statement.) See "Error Handling" in the *Programmer's Guide* for more information on how to trap run-time errors that have an **ERR** code.

Table 4.1 lists the error codes in numerical order. See the alphabetical listing for explanations of the errors. Note that a few error messages in the alphabetical listing include a sentence like the following:

"In the QBX environment, an ERROR 37 statement will generate this message."

This notation means that you should, in general, avoid using the ERROR 37 statement in a program that will be run in the QBX environment. Such errors are *not* trappable run-time errors.

Table 4.1 Run-Time Error Codes

Code	Message	Code	Message
1	NEXT without FOR	52	Bad filename or number
2	Syntax error	53	File not found
3	RETURN without GOSUB	54	Bad file mode
4	Out of DATA	55	File already open
5	Illegal function call	56	FIELD statement active
6	Overflow	57	Device I/O error
7	Out of memory	58	File already exists
8	Label not defined	59	Bad record length
9	Subscript out of range	61	Disk full
10	Duplicate definition	62	Input past end of file
11	Division by zero	63	Bad record number
12	Illegal in direct mode	64	Bad filename
13	Type mismatch	67	Too many files
14	Out of string space	68	Device unavailable
16	String formula too complex	69	Communication-buffer overflow
17	Cannot continue	70	Permission denied
18	Function not defined	71	Disk not ready
19	No RESUME	72	Disk-media error
20	RESUME without error	73	Feature unavailable
24	Device timeout	74	Rename across disks
25	Device fault	75	Path/File access error
26	FOR without NEXT	76	Path not found
27	Out of paper	80	Feature removed
29	WHILE without WEND	81	Invalid name
30	WEND without WHILE	82	Table not found
33	Duplicate LABEL	83	Index not found
35	Subprogram not defined	84	Invalid column
37	Argument-count mismatch	85	No current record
38	Array not defined	86	Duplicate value for unique index
40	Variable required	87	Invalid operation on null index
50	FIELD overflow	88	Database needs repair
51	Internal error		

Invocation, Compile-Time, and Run-Time Error Messages

Argument-count mismatch

An incorrect number of arguments was used in a **SUB** or **FUNCTION** procedure call.

Compare the **DECLARE** statement for the **SUB** or **FUNCTION** procedure with the **CALL** statement to make sure the argument list has the same number of items in both cases. In the QBX environment, you can use the Search menu's Find command to look at the **DECLARE** statement.

In the QBX environment, an ERROR 37 statement will generate this message.

(Compile-time error)

Array already dimensioned

This error can be caused by any of the following:

- More than one **DIM** statement for the same static array.
- An attempt to redimension a dynamic array with a **DIM** statement without using the **ERASE** statement to deallocate the array first.
- An **OPTION BASE** statement that occurs after an array is dimensioned.

(Compile-time or run-time error)

Array not defined

Your program is attempting to use an array that is not currently defined.

Make sure the array is defined with a **DIM** or **REDIM** statement.

In the QBX environment, an ERROR 38 statement will generate this message.

(Compile-time error)

Array not dimensioned

An array is referred to but its dimensions haven't been declared.

If you are compiling with BC, this error is not "fatal"; that is, the program will execute, although program results may be incorrect.

(BC compile-time warning)

Array too big

There is not enough user data space to accommodate the array declaration. Try:

- Reducing the size of the array, or
- Using the **$DYNAMIC** metacommand at the beginning of your program.

This error also may occur if the array size exceeds 64K, the array is not dynamic, and the /AH option was not used on the compiler invocation line. (All three of these conditions must be present to generate the error.) Try:

- Reducing the size of the array, or
- Making the array dynamic and using the /AH command-line option.

(BC compile-time error)

AS clause required

A variable declared with an **AS** clause is referred to without one.

If the first declaration of a variable has an **AS** clause, every subsequent **DIM**, **REDIM**, **SHARED**, or **COMMON** statement that refers to that variable must have an **AS** clause.

(Compile-time error)

AS clause required on first declaration

A variable that has not been declared using an **AS** clause is referred to with an **AS** clause.

(Compile-time error)

AS missing

The compiler expects an **AS** keyword, as in the following correct statement:

```
OPEN "FILENAME" FOR INPUT AS #1
```

(BC compile-time error)

Asterisk missing

The asterisk is missing from a string definition in a user-defined type.

(Compile-time error)

Bad file mode

Example causes of this error are:

- A **PUT** or **GET** statement specified a sequential file.
- A **FIELD** statement specified a file not opened for random access.
- A **PRINT#** statement specified a sequential file opened for input.
- An **INPUT#** statement specified a file opened for output or appending.

- BASIC tried to use an include file previously saved in compressed format. Include files must be saved in text format. Reload the include file, save it in text format, then try to run the program again.
- An **OPEN FOR ISAM** statement used a file-number argument that is already being used by an open file.
- An ISAM statement or function file-number argument refers to an open file that is not an ISAM table.
- A filename used in the ISAM **DELETE** statement does not refer to a file.

(Run-time error)

ERR code: 54

Bad file name

An filename that does not follow the appropriate naming convention is used with **OPEN**, **KILL**, **SAVE**, or **LOAD** (for example, the filename has too many characters).

(Run-time error)

ERR code: 64

Bad file name or number

This error may occur because of one of these situations:

- A statement or command refers to a file with a file number or name that is not specified in the **OPEN** statement or is out of the range of file numbers specified earlier in the program.
- No ISAM table (or any other file) is currently open with the file number used in an ISAM statement or function.
- A file exists with the filename used in an **OPEN FOR ISAM** statement, but it was not created by ISAM.
- You have used an invalid filename in an **OPEN FOR ISAM** statement. A filename must conform to the DOS-file naming rules; it may contain a drive letter and a path.

(Run-time error)

ERR code: 52

Bad record length

A **GET** or **PUT** statement was executed that specified a record variable whose length did not match the record length specified in the corresponding **OPEN** statement.

(Run-time error)

ERR code: 59

Bad record number

The record number in a **GET** or **PUT** statement was less than or equal to zero.

(Run-time error)

ERR code: 63

BASE missing

BASIC expected the keyword **BASE** here, as in **OPTION BASE**.

(BC compile-time error)

Binary source file

This message can be generated when:

- The file you have attempted to compile is not compatible with this version of BASIC. Try saving the binary file as an ASCII file and then recompile it. (All source files saved by BASICA should be saved with the , A option.)
- You try to use the /ZI or /ZD CodeView options with binary source files. (This is a warning message only.)

(Compile-time error or invocation warning)

Block IF without END IF

There is no corresponding **END IF** in a block **IF** construct.

(Compile-time error)

Breakpoints not allowed on CASE clauses or END SELECT

There are some restrictions on using breakpoints to debug **SELECT...END SELECT** statements. You cannot place a breakpoint on the first statement in the **CASE** block or after the **END SELECT** statement.

To debug a **SELECT...END SELECT** statement with the aid of a breakpoint, set the breakpoint on the **SELECT CASE** line in your program. Then single-step through the rest of the statement structure.

(QBX run-time error)

Buffer size expected after /C:

You must specify a buffer size after the /C option.

(BC invocation error)

BYVAL allowed only with numeric arguments

You cannot pass string or record arguments with a **BYVAL** clause.

(BC compile-time error)

/C: buffer size too large

The maximum size of the communications buffer is 32,767 bytes.

(BC invocation error)

Cannot continue

While debugging, you have made a change that prevents execution from continuing.

In the QBX environment, `Cannot continue` is a prompt that may occur while you are debugging. While your program was suspended (at a breakpoint, for example) you made a change to the program that has implications throughout the program, not just the point where you made that change. You may have redimensioned an array, changed procedure arguments, or edited some other declarative statement. Use the dialog box and either

- Choose OK to keep the change and restart the program, or
- Choose Cancel to undo the change in your program text. (You may do this if restarting interferes with your debugging strategy.)

In the QBX environment, an ERROR 17 statement will generate this message.

(Run-time error and QBX prompt)

Cannot find file (*filename*). Input path:

This error occurs when BASIC cannot find a Quick library or utility (BC.EXE, LINK.EXE, LIB.EXE, or QBX.EXE) required by the program.

Enter the correct path, or press Ctrl+C to return to the system prompt.

(QBX invocation error)

Cannot find file (*filename*)—Disk I/O error

This error is caused by physical problems accessing the disk; for example, the drive door is left open.

(QBX invocation error)

Cannot generate listing for BASIC binary source files

You are attempting to compile a binary source file with the BC command and the /A option. Recompile without the /A option.

(BC invocation error)

Cannot start with 'FN'

You used "FN" as the first two letters of a subprogram or variable name. "FN" can only be used as the first two letters when calling a **DEF FN** function.

(QBX compile-time error)

CASE without SELECT

The first part of a **SELECT CASE** statement is missing or misspelled.

Also check other structures within the **SELECT...END SELECT** structure and verify that they are correctly matched. For example, an **IF** block without a matching **END IF** inside the **SELECT...END SELECT** structure generates this error.

(Compile-time error)

Colon expected after /C

A colon is required between the option and the buffer-size argument.

(BC invocation error)

Comma missing

BASIC expects a comma.

(Compile-time error)

COMMON and DECLARE must precede executable statements

A **COMMON** statement or a **DECLARE** statement is misplaced. **COMMON** and **DECLARE** statements must appear before any executable statements. All BASIC statements are executable except the following:

- **COMMON**
- **CONST**
- **DATA**
- **DECLARE**
- **DEF***type*
- **DIM** (for static arrays)
- **OPTION BASE**
- **REM**
- **SHARED**
- **STATIC**
- **TYPE**
- All metacommands

(Compile-time error)

COMMON in Quick library too small

More common variables are specified in the module than in the currently loaded Quick library.

(QBX compile-time error)

COMMON name illegal

BASIC encountered an illegal block-name specification (for example, a block name that is a BASIC reserved word) in a named **COMMON** block.

(BC compile-time error)

Communication-buffer overflow

During remote communications, the receive buffer overflowed.

The size of the receive buffer is set by the /C command-line option or the RB option in the **OPEN COM** statement. To avoid this error, you can:

- Check the buffer more frequently (with the **LOC** function).
- Empty the buffer more often (with the **INPUT$** function).

(Run-time error)

ERR code: 69

CONST/DIM SHARED follows SUB/FUNCTION

CONST and **DIM SHARED** statements should appear before any **SUB** or **FUNCTION** procedure definitions.

If you are compiling with BC, this error is not "fatal"; that is, the program will execute, although the results may be incorrect.

(Compile-time warning)

Control structure in IF...THEN...ELSE incomplete

An unmatched **NEXT**, **WEND**, **END IF**, **END SELECT**, or **LOOP** statement appears in a single-line **IF...THEN...ELSE** statement.

(BC compile-time error)

Currency type illegal in alternate math pack

This error occurs whenever the currency type is used with an alternate math pack. If you use the /Fpa option while compiling, your program cannot contain:

- The at symbol (@) to denote currency data type.
- The **DEFCUR** statement.
- The **CURRENCY** keyword.

(BC compile-time error)

Database needs repair

An **OPEN FOR ISAM** statement attempted to open a file that is in need of repair. You may want to use the REPAIR.EXE utility to recover (restore physical integrity to) the database.

(Run-time error)

ERR code: 88

Data-memory overflow

There is too much program data to fit in memory. This error is often caused by too many constants or too much static array data. If you are using the BC command, or the Make EXE File or Make Library command, try turning off any debugging options. If memory is still exhausted, break your program into multiple modules or use the /AH and /FS options.

(Compile-time error)

DECLARE required

An implicit **SUB** or **FUNCTION** procedure call appears before the procedure definition. (An implicit call does not use the **CALL** statement.) All procedures must be defined or declared before they are implicitly called.

(Compile-time error)

DEF FN not allowed in control statements

DEF FN function definitions are not permitted inside control constructs such as **SELECT CASE** and **IF...THEN...ELSE**.

(QBX compile-time error)

DEF without END DEF

There is no corresponding **END DEF** in a multiline function definition.

(Compile-time error)

DEFtype character specification illegal

A **DEF***type* statement is entered incorrectly. **DEF** can be followed only by **LNG**, **DBL**, **INT**, **SNG**, **CUR**, **STR**, or (for user-defined functions) a blank space.

(BC compile-time error)

Device fault

A device has returned a hardware error, which can indicate one of the following conditions:

- If you are attempting to print a file, the printer is not attached to the parallel port LPT1.
- If data is being transmitted to a communications file, the signals being tested with the **OPEN COM** statement were not found in the specified period of time.

In the QBX environment, this message can mean that QBX has detected a fault at the printer. Make sure that:

- The printer cable is securely connected to both the printer and the computer.
- The printer power is still on.
- The printer online indicator light is still on.

(Run-time error)

ERR code: 25

Device I/O error

An input or output error occurred while your program was using a device, such as the printer or disk drive. The operating system cannot recover from the error.

In the QBX environment, this message can occur only when you are using the File menu's Print command. The printer has malfunctioned.

(Run-time error)

ERR code: 57

Device timeout

The program did not receive information from an I/O device within a predetermined amount of time.

In the QBX environment, while attempting to execute the Print command from the File menu, QBX tried repeatedly to send text to a printer attached to the LPT1 port, without getting an acknowledging signal back from the printer. Make sure that there is a printer attached to the LPT1 port.

(Run-time error)

ERR code: 24

Device unavailable

The device you are attempting to access is not online or does not exist.

In the QBX environment, this message can mean that you have attempted to open a file on a device that does not exist in your system. Check the list of available device names in the Dirs/Drives list box and use one of those.

(Run-time error)

ERR code: 68

Disk full

There wasn't enough room on the disk for the completion of a **PRINT#**, **WRITE#**, or **CLOSE** operation. This error can also occur if there is not enough room for QBX to write out an object or executable file.

In the QBX environment, this message can mean that there isn't room on the specified disk to save a specified file. You may:

- Save the file to another disk.
- Cancel the command and use the QBX Shell command from the File menu to delete some files from this disk. Then try this command again.

(Run-time error)

ERR code: 61

Disk-media error

Disk-drive hardware has detected a physical flaw on the disk.

In the QBX environment, this can mean that QBX does not recognize the format of the disk it is attempting to use. Try a different disk or reformat the current disk.

(Run-time error)

ERR code: 72

Disk not ready

The disk-drive door is open, or no disk is in the drive.

In the QBX environment, this can mean that there is no disk in the drive specified in a dialog box. Insert a disk in the drive and retry the operation.

(Run-time error)

ERR code: 71

Division by zero

This error occurs when you divide by zero in an expression.

(Compile-time or run-time error)

ERR code: 11

DO without LOOP

The terminating **LOOP** clause is missing from a **DO...LOOP** statement.

(Compile-time error)

Document too large

QBX cannot load a document file larger than 64K.

Use another editor to divide your document into separate files, none of which exceed 64K.

(QBX run-time error)

Duplicate definition

This error occurs if you are attempting to create a new element in your program and have given it a name that is already being used.

For example:

- A **CONST** statement uses the same name as an existing variable.
- A new variable or procedure has the same name as an existing procedure.

This error also can occur if you attempt to create a file using the name of a file that is already loaded. Use a different filename in the dialog box or cancel the command and rename or unload the conflicting file.

Starting a variable name with the letters "FN" is another way to generate this error.

For ISAM, trying to create an index that already exists will cause this error.

This error also occurs if you attempt to change the dimensions of a static array while your program is running. If you plan to change the dimensions of an array while your program is running, make sure it is a dynamic array and use the **REDIM** statement to make the change.

(Compile-time or run-time error)

ERR code: 10

Duplicate label

Two program lines were assigned the same number or label. Each line number or label in a module must be unique.

Check all of the **SUB** and **FUNCTION** procedure definitions in the module, as well as the module-level code, for the duplicate label. In the QBX editor, the Search menu's Find command is handy for this.

In the QBX environment, an ERROR 33 statement will generate this message.

(Compile-time error)

Duplicate value for unique index

Either of these conditions can generate this error:

- A unique index is current for an ISAM table, and the user has attempted to enter a value for a column that is already in the column.
- A **CREATEINDEX** statement has been executed with a non-zero *unique* argument value, but the existing column(s) already contained non-unique values.

(Run-time error)

ERR code: 86

Dynamic array element illegal

Dynamic array elements are not allowed with **VARPTR$**.

(BC compile-time error)

Dynamic array illegal

Dynamic arrays cannot be user-defined type elements. If you use an array as an element of a user-defined type, it must be a static array.

(BC compile-time error)

Element not defined

A user-defined type element is referred to but not defined. For example, if the user-defined type `MYTYPE` contained elements `A`, `B`, and `C`, then an attempt to use the variable `D` as an element of `MYTYPE` would cause this message to appear.

(Compile-time error)

ELSE without IF

An **ELSE** clause appears without a corresponding **IF**. Sometimes this error is caused by incorrectly nested **IF** statements.

Also check other control structures within the **IF...END IF** block and verify that they are correctly matched. For example, a nested **IF** block without a matching **END IF** inside the outer **IF...END IF** block generates this error.

(Compile-time error)

ELSEIF without IF

An **ELSEIF** statement appears without a corresponding **IF**. Sometimes this error is caused by incorrectly nested **IF** statements.

(BC compile-time error)

EMS corrupt

The BASIC overlay manager cannot complete a task because of an EMM (Expanded Memory Manager) error or a conflict between programs competing for expanded memory. This is a fatal run-time error.

To remedy this error, you can either:

- Link your program with NOEMS.OBJ, or
- Fix the problem at the expanded-memory configuration level.

(Run-time error)

END DEF without DEF

An **END DEF** statement has no corresponding **DEF** statement.

Also check other control structures within the **DEF...END DEF** structure and verify that they are correctly matched. For example, an **IF** without a matching **END IF** inside the **DEF...END DEF** structure generates this error.

(Compile-time error)

END IF without block IF

The beginning of an **IF** block is missing.

Also check other control structures within the **IF...END IF** block and verify that they are correctly matched. For example, a nested **IF** block without a matching **END IF** inside the outer **IF...END IF** block generates this error.

(Compile-time error)

End of file unexpected in TYPE declaration

If, while parsing a **TYPE** statement, the compiler reaches the end of the source file without seeing the **END TYPE** statement, the program halts execution.

(BC compile-time error)

END SELECT without SELECT

The end of a **SELECT CASE** statement appears without a beginning **SELECT CASE**. The beginning of the **SELECT CASE** statement may be missing or misspelled.

Also check other control structures within the **SELECT...END SELECT** structure and verify that they are correctly matched. For example, an **IF** block without a matching **END IF** inside the **SELECT...END SELECT** structure generates this error.

(Compile-time error)

END SUB or END FUNCTION must be last line in window

You are attempting to add code after the last line in a procedure. You must either return to the main module or open another module.

(QBX compile-time error)

END SUB/FUNCTION without SUB/FUNCTION

The **SUB** or **FUNCTION** statement is missing from a procedure. You may have deleted it during editing.

(Compile-time error)

END TYPE without TYPE

An **END TYPE** statement is used outside a **TYPE** declaration.

(QBX compile-time error)

Equal sign missing

BASIC expects an equal sign.

(Compile-time error)

Error during QBX initialization

Several conditions can cause this error. It is most commonly caused when there is not enough memory in the machine to load BASIC. If you are loading a Quick library, try reducing the size of the library.

This error may occur when you attempt to use BASIC on unsupported hardware.

(QBX invocation error)

Error in loading file (filename)—Cannot find file

This error occurs when loading a Quick library with redirecting input and/or output to BASIC. The input file is not at the location specified on the command line.

(QBX invocation error)

Error in loading file (filename)—Disk I/O error

This error is caused by physical problems accessing the disk; for example, the drive door is left open.

(QBX invocation error)

Error in loading file (filename)—DOS memory-arena error

The area of memory used by DOS has been written to, either by an assembly-language routine or with the **POKE** statement.

(QBX invocation error)

Error in loading file (filename)—Invalid format

You are attempting to load a Quick library that is not in the correct format. This may be caused by one of the following conditions:

- You are attempting to use a Quick library created with a previous version of BASIC.
- You are trying to use a file that has not been processed with BASIC's Make Library command or the /QU option from LINK.
- You are trying to load a stand-alone (.LIB) library with BASIC.

(QBX invocation error)

Error in loading file (filename)—Out of memory

More memory was required than is available. For example, there may not be enough memory to allocate a file buffer.

In DOS:

- Reduce the size of your DOS buffers
- Eliminate any terminate-and-stay-resident programs.
- Eliminate some device drivers.

In BASIC:

- Place a **$DYNAMIC** metacommand at the top of your program if you have large arrays.
- Make your programs smaller.
- In the QBX environment, unload any document or source files that are in memory but not needed.
- Use the /FS or /AH option.

(QBX invocation error)

Error in loading run-time module module-name: Cannot find file in PATH

This error occurs when loading a run-time module with redirected input and/or output. The named run-time module is not on the path specified in the DOS PATH environmental variable.

(Run-time error)

Error in loading run-time module module-name: Disk I/O error

This error is caused by physical problems accessing the disk.

(Run-time error)

Error in loading run-time module module-name: DOS memory-arena error

The arena of memory used by DOS has been written to, either by a program bug in an assembly-language or C routine, or with the **POKE** statement.

(Run-time error)

Error in loading run-time module *module-name*: Incompatible run-time module

Either one of the following conditions can generate this error:

- There is a mismatch between common blocks in your application and the custom run-time module. If a routine embedded in a custom run-time module defines named or unnamed common blocks, then routines in your application must not attempt to define those blocks as **COMMON** with a larger size.
- You have mistakenly named a file with the same name as one of your run-time modules. Either the mistakenly named file is not a run-time module or it is a run-time module built for a different target environment than the one currently running.

(Run-time error)

Error in loading run-time module *module-name*: Invalid format

You are attempting to load a run-time module that is not in the correct format.

(Run-time error)

Error in loading run-time module *module-name*: Memory allocation error

This is a fatal internal error. You will have to rebuild your memory allocation from scratch by, for example, rebooting the system.

(Run-time error)

Error in loading run-time module *module-name*: Out of memory

More memory was required to load the run-time module than is available.

To get more memory for your run-time module using DOS:

- Reduce the size of your DOS buffers.
- Eliminate any terminate-and-stay-resident programs.
- Eliminate some device drivers.

Use BASIC to:

- Place a **$DYNAMIC** metacommand at the top of your program if you have large arrays.
- Make your programs smaller.

(Run-time error)

Error loading overlay

You must relink or, in some cases, recompile the modules in order to get a valid overlay. This is a fatal run-time error.

(Run-time error)

/Es option incompatible with /Ea

You cannot use the /Es option with the /Ea option. You can use one or the other but not both.

(Invocation error)

EVENT ON without /v or /w on command line

The **EVENT ON** statement requires a /V or /W option on the command line.

This is a warning message only, and can be generated in two different situations:

- If there are **ON** *event* statements (such as **ON KEY** or **ON COM**) in the program, you must use a /V or /W option on the BC command line in order for the program to run properly.
- If there are no **ON** *event* statements in the program, then you must change the **EVENT ON** statement to an **EVENT OFF** statement, and must not use a /V or /W option.

(Compile-time warning)

EXIT DO not within DO...LOOP

An **EXIT DO** statement is used outside of a **DO...LOOP** statement.

(Compile-time error)

EXIT FOR not within FOR...NEXT

An **EXIT FOR** statement is used outside of a **FOR...NEXT** statement.

(Compile-time error)

Expected: *item*

This is a syntax error. The cursor is positioned at the unexpected item. Examples of items that may appear after `Expected:` are expression, variable, identifier, statement, label, parameter, end-of-statement, or string constant, **BYVAL**, **SEG**, a right parenthesis, or the operators <, <=, >, >=, =, or <>.

(QBX compile-time error)

Expression too complex

Certain internal limitations of BASIC have been exceeded. For example, during expression evaluation, strings that are not associated with variables are assigned temporary locations. A large number of such strings can cause this error to occur. Likewise, a numeric expression with many complicated subexpressions can cause this error.

Try simplifying expressions or assigning strings to variables.

(Compile-time error)

Extra file name ignored

You specified too many files on the command line; the last filename on the line is ignored.

(BC invocation warning)

Far heap corrupt

The far-heap memory has been corrupted by one of the following:

- The **POKE** statement modified areas of memory used by BASIC. (This may modify the descriptor for a dynamic array of numbers or fixed-length strings.)
- The program called an external routine that modified areas of memory used by BASIC. (This may modify the descriptor for a dynamic array of numbers or fixed-length strings.)

(Run-time error)

Feature removed

This feature is not available because you removed if by linking with a stub file or you explicitly chose to eliminate it with the /NoFrills command-line option.(Run-time error)

ERR code: 80

Feature unavailable

You may be attempting to use a feature of another version of BASIC that is not available with Microsoft BASIC.

Or, you may be attempting to use a feature of Microsoft BASIC that is unavailable under the operating system you are running. Examples of this are:

- Using the **LOCK** or **UNLOCK** statement while running DOS 2.1.
- Using the **SHELL** function while running DOS instead of OS/2.
- Using the **PALETTE** or **PCOPY** statement while running OS/2.

(Compile-time or run-time error)

ERR code: 73

FIELD overflow

A **FIELD** statement attempted to allocate more bytes than were specified for the record length of a random file.

(Run-time error)

ERR code: 50

FIELD statement active

A **GET** or **PUT** statement specified a record-variable on a file for which **FIELD** statements had been executed.

GET or **PUT** with a record variable argument may be used only on files where no **FIELD** statements have been executed.

(Run-time error)

ERR code: 56

File already exists

The filename specified in a **NAME** statement is identical to a filename that is already in use on the disk.

(Run-time error)

ERR code: 58

File already open

This error is caused by one of the following conditions:

- A sequential-output-mode **OPEN** statement was executed for a file that is already open.
- A **KILL** statement refers to a file that is already open.
- The file named in an **OPEN FOR ISAM** statement is already open for ISAM.
- A file with the same name as the file argument in an **OPEN FOR ISAM** statement is already open for another access mode.

(Run-time error)

ERR code: 55

File not found [[in module *module-name* at address *segment:offset*]]

A **FILES**, **KILL**, **NAME**, **OPEN**, or **RUN** statement refers to a file that does not exist.

The form of the error message that includes a module name and address occurs during execution of compiled programs. The variable *module-name* is the name of the calling module. The address is the location of the error in the code.

(Run-time error)

ERR code: 53

File previously loaded

You are attempting to load a file that is already in memory. You may:

- Work with the file already loaded in memory.
- Unload the file that is already in memory so you can load this one.
- Look at the list of files available for loading. You may have tried to load the wrong one.

(QBX compile-time error)

Fixed-length string illegal

You can't use a fixed-length string as a formal parameter to a **SUB** or **FUNCTION** procedure.

(Compile-time error)

FOR index variable already in use

This error occurs when an index variable is used more than once in nested **FOR** loops.

(BC compile-time error)

FOR index variable illegal

This error is usually caused when an incorrect variable type is used in a **FOR**-loop index. A **FOR**-loop index variable must be a simple numeric variable.

(BC compile-time error)

FOR without NEXT

Each **FOR** statement must have a matching **NEXT** statement.

In the QBX environment, an ERROR 26 statement will generate this message.

(Compile-time error)

Formal parameter specification illegal

There is an error in a **SUB** or **FUNCTION** parameter list.

(BC compile-time error)

Formal parameters not unique

A **SUB** or **FUNCTION** declaration contains duplicate parameters, as in this example:

```
SUB GetName(A,B,C,A) STATIC
```

(BC compile-time error)

Function already defined

This error occurs when a previously defined **SUB** or **FUNCTION** procedure or **DEF FN** function is redefined.

(BC compile-time error)

Function name illegal

The name of a **DEF FN** function must begin with **FN**.

(BC compile-time error)

Function not defined

You must define a **DEF FN** function before using it.
In the QBX environment, the module or Quick library that contains the function definition may not be loaded.

In the QBX environment, an ERROR 18 statement will generate this message.

(Compile-time or run-time error)

GOSUB missing

GOSUB is missing from an **ON** *event* statement.

(BC compile-time error)

GOTO missing

GOTO is missing from an **ON ERROR** statement.

(BC compile-time error)

GOTO or GOSUB expected

BASIC expects a **GOTO** or **GOSUB** statement.

(BC compile-time error)

Identifier cannot end with %, &, !, #, $, or @

The above suffixes are not allowed in type identifiers, **SUB** procedure names, or names appearing in **COMMON** statements.

(QBX compile-time error)

Identifier cannot include period

User-defined type identifier and record element names cannot contain periods. A variable name cannot contain a period if the part of the name before the period has been used in an **AS** *usertype* clause anywhere in the program.

Although variable names can contain periods, it is recommended that a period be used only as a record variable separator. If you have programs that use the period in variable names, you could change them to use mixed case instead. For example, variable `ALPHA.BETA` would become `AlphaBeta`.

(Compile-time error)

Identifier expected

You are attempting to use a number or a BASIC reserved word where an identifier is expected.

(BC compile-time error)

Identifier too long

Identifiers must not be longer than 40 characters.

(QBX compile-time error)

Illegal function call

You are attempting to give an improper or out-of-range argument to a BASIC statement. Examples of such errors are:

- A negative or unreasonably large subscript is used.
- A negative number is raised to a power that is not an integer.
- A negative record number is given when using **GET** or **PUT**.
- A **BLOAD** or **BSAVE** operation is directed to a nondisk device.
- An I/O function or statement (**LOC** or **LOF**, for example) is performed on a device that does not support it.
- Strings are concatenated to create a string greater than 32,767 characters in length.
- An **ISAM BEGINTRANS** statement has been executed while a transaction is already pending.
- An **ISAM COMMITTRANS** statement has been executed without a transaction pending.
- An **ISAM ROLLBACK** statement has referred to a savepoint that does not exist or a **ROLLBACK ALL** statement has been executed with no transaction pending.

(Run-time error)

ERR code: 5

Illegal in direct mode

In the QBX environment, the highlighted statement is valid only within a program and cannot be used in the Immediate window. In general, the following statements and metacommands cannot be used in the Immediate window:

- **COMMON**, **CONST**, **DATA** , **DECLARE**, **DIM**, **OPTION**, **SHARED**, **STATIC**, and **TYPE** nonexecutable statements
- **$INCLUDE**, **$DYNAMIC**, and **$STATIC** metacommands
- **DEF FN...END DEF**, **ELSE IF**, **END IF**, **END TYPE**, **FUNCTION...END FUNCTION**, **REDIM**, and **SUB...END SUB** statements

In the QBX environment, an ERROR 12 statement will generate this message.

(QBX compile-time error)

Illegal in SUB, FUNCTION, or DEF FN

The compiler has encountered a statement that is allowed only in module-level code. For example, **CLEAR** and **STACK** statements are allowed only in module-level code.

(Compile-time error)

Illegal number

You have used a number inappropriate for the context in which it is used. For example, BASIC doesn't allow you to **DIM** a fixed-length string of zero-length, so DIM X as STRING * 0 is illegal. An illegal number also occurs if you declare contradictory values, e.g., 10.52%.

(QBX compile-time error)

Illegal outside of SUB, FUNCTION, or DEF FN

The compiler has encountered a statement that is not allowed in module-level code. For example, **EXIT SUB**, **EXIT FUNCTION**, or **EXIT DEF** statements are not allowed in module-level code.

(Compile-time error)

Illegal outside of SUB or FUNCTION

The compiler has encountered a statement that is not allowed in module-level code or **DEF FN** functions.

(Compile-time error)

Illegal outside of TYPE block

The *element* AS *type* clause is permitted only within a **TYPE...END TYPE** block.

(QBX compile-time error)

Illegal type character in numeric constant

A numeric constant contains an inappropriate type-declaration character.

(BC compile-time error)

$INCLUDE-file access error

The include file named in the **$INCLUDE** metacommand cannot be located.

(BC compile-time error)

Include file too large

QBX cannot load an include file larger than 64K.

Use a different editor to divide your include file into separate files, none of which exceed 64K.

(QBX compile-time error)

Incorrect DOS version. Link with OVLDOS21.OBJ

An overlay could not be found in the current directory or on the current path while you are running under DOS 2.1. The BASIC run-time overlay manager requires the program OVLDOS21.OBJ to be linked with your program in order to successfully search along the path under DOS 2.1. This is a fatal run-time error.

(Run-time error)

Index not found

There is no index with the specified name for the ISAM table referred to by a file number argument (for example, in a **DELETEINDEX** or **SETINDEX** statement).

(Run-time error)

ERR code: 83

Input file not found

The source file you named on the BC command line was not found, so the compilation was not carried out.

Check on the spelling of the source filename and the subdirectory location of the source file. Then retype the BC command line with the correct path specification for the source file.

(BC invocation error)

INPUT missing

BASIC expects the reserved word **INPUT**.

(BC compile-time error)

Input past end of file

An **INPUT #** statement reads from a null (empty) file or from a file in which all data have already been read.

To avoid this error, use the **EOF** function to detect the end of file.

(Run-time error)

ERR code: 62

Input path for run-time module module-name:

This prompt appears if the run-time module is not found. Enter the correct path specification at the prompt in order to continue running the program.

(Run-time prompt)

Insufficient EMS to load overlays

If all of your overlays are smaller than 16K, you will never see this message. If you do see this message, you can do one of the following:

- Reconfigure EMS so more of it is available for BASIC overlays. For example, reduce the size of the disk cache or RAM disk. The maximum amount needed for BASIC overlays is 64K for each overlay specified in the **LINK** command.
- Link with NOEMS.OBJ to force overlays to be loaded from disk. Your program will run slower if you do this.

(Run-time error)

Integer between 1 and 32767 required

The statement requires an integer argument in the specified range.

(BC compile-time error)

Internal error

An internal malfunction occurred in BASIC. Use the Product Assistance Request form included with your documentation to report to Microsoft the conditions under which the message appeared.

(Run-time error)

ERR code: 51

Internal error near xxxx

An internal malfunction occurred in BASIC at the offset-address location *xxxx*. Use the Product Assistance Request form included with your documentation to report the conditions under which the message appeared.

(BC compile-time error)

Invalid character

BASIC found an invalid character, such as a control character, in the source file.

(BC compile-time error)

Invalid column

When you open a file for ISAM, you cannot specify a data type that includes an element name that is not a column name in the file.

When you create an ISAM index, you cannot name a column that:

- Does not exist, or
- Has a data type that cannot be indexed (an array or user-defined type).

(QBX run-time error)

ERR code: 84

Invalid constant

An invalid expression has been used to assign a value to a symbolic constant.

Numeric expressions assigned to symbolic constants may contain:

- Numeric literals.
- Previously defined symbolic constants.
- Any of the arithmetic or logical operators except exponentiation.

String expressions assigned to a symbolic constant may consist only of a single literal string, enclosed in double quotation marks.

(QBX compile-time error)

Invalid identifier

You've used a character not valid in an identifier. You may have used a character that isn't a blank or A-Z in a **DEF FN** name, or entered a character that isn't valid in a procedure name using either the Edit New FUNCTION or Edit New SUB dialog box. Beginning a procedure name with a digit is not allowed, nor is a using an identifier that contains characters other than letters and numbers or any of the type definition suffixes (%, &, !, #, @, or $).

(QBX compile-time error)

Invalid name

The rules for naming an ISAM object (such as a table, column, or index) are different than the rules for naming a BASIC object such as a variable or procedure. ISAM names:

- Must be 30 characters or less.
- May contain only the characters A-Z, a-z, and 0-9.

(Run-time error)

ERR code: 81

Invalid operation on NULL index

You cannot perform an ISAM **MOVELAST** or **SEEK** operation on a NULL index.

(QBX run-time error)

ERR code: 87

Label not defined

A line label is referred to (in a **GOTO** statement, for example), but does not occur in the program. In the QBX environment, an ERROR 8 statement generates this message.

(Compile-time error)

ERR code:8

Label not defined: *label*

A **GOTO** *linelabel* statement refers to a nonexistent line label.

(BC compile-time error)

Left parenthesis missing

BASIC expected a left parenthesis, or a **REDIM** statement tried to reallocate space for a scalar.

(BC compile-time error)

Line invalid. Start again

An invalid filename character was used following the backslash (\) or colon (:) path characters.

(BC invocation error)

Line number or label missing

A line number or label is missing from a statement that requires one. For example, a **GOTO** statement requires a line number or label argument.

(BC compile-time error)

Line too long

Lines are limited to 255 characters.

(BC compile-time error)

LOOP without DO

The **DO** starting a **DO...LOOP** statement is missing or misspelled.

Also check other control structures within the **DO...LOOP** structure and verify that they are correctly matched. For example, an **IF** block without a matching **END IF** inside the **DO...LOOP** structure generates this error.

(Compile-time error)

Lower bound exceeds upper bound

The lower bound exceeds the upper bound defined in a **DIM** statement.

(BC compile-time error)

Math overflow

The result of a calculation is too large to be represented in BASIC number format.

(BC compile-time error)

$Metacommand error

The syntax of a metacommand is incorrect.

Metacommands must be preceded by a comment. For example, both of the **$INCLUDE** metacommands below are correct:

```
REM  $INCLUDE:  'filespec'
'  $INCLUDE:  'filespec'
```

If you are compiling the program with BC this error is not "fatal"; that is, the program will execute, although the results may be incorrect.

(BC compile-time warning or QBX compile-time error)

Minus sign missing

BASIC expects a minus sign.

(BC compile-time error)

Module level code too large

The size of your module-level code exceeds QBX's internal limit. Try moving some of the code into **SUB** or **FUNCTION** procedures.

(QBX compile-time error)

Module not found. Unload module from program?

When QBX loaded a multiple-module program, it could not find one of the modules listed in the .MAK file. To complete the load process, QBX created an empty module to satisfy the .MAK file's module list.

When you run the program, you may do one of the following:

- Choose OK to delete the empty module and the .MAK file entry for the module.
- Choose Cancel to cancel this run operation so you can either reconcile the .MAK file entry with the module file you may have in another directory, or save the program text you have just entered into the empty module so you can try running it again.

(QBX run-time error)

Must be first statement on the line

In block **IF...THEN...ELSE** constructs, only a line number or label may precede **IF**, **ELSE**, **ELSEIF**, and **END IF**.

In **SELECT...END SELECT** constructs, only a line number or label may precede **CASE** and **END SELECT**.

(Compile-time error)

Name of subprogram illegal

A **SUB** or **FUNCTION** procedure name is a BASIC reserved word, or the same name appears as a variable or **SUB** or **FUNCTION** name earlier in the module.

(BC compile-time error)

Nested function definition

A **SUB** or **FUNCTION** procedure or a **DEF FN** function appears inside another **SUB** or **FUNCTION** procedure or **DEF FN** function.

(BC compile-time error)

NEXT missing for *variable*

A **FOR** statement is missing a corresponding **NEXT** statement. The *variable* is the **FOR**-loop index variable.

(BC compile-time error)

NEXT without FOR

Each **NEXT** statement must have a matching **FOR** statement.

In the QBX environment, an ERROR 1 statement will generate this error.

(Compile-time error)

No current record

There was no current record to delete, retrieve, or update when an ISAM **DELETE**, **RETRIEVE**, or **UPDATE** statement was executed.

An ISAM **MOVENEXT** or **MOVEPREVIOUS** statement has been executed on a file where there is no next or previous record.

(Run-time error)

ERR code: 85

No line number in *module-name* **at address** *segment:offset*

This error occurs when the error address cannot be found in the line-number table during error trapping. This happens if there are no integer line numbers between 0 and 65,527 in the program. It may also occur if the line-number table has been accidentally overwritten by the user program. This error is severe and cannot be trapped.

(Run-time error)

No main module. Choose Set Main Module from the Run menu to select one

You are attempting to run the program after you have unloaded the main module. Every program must have a main module so QBX knows where to start running the program.

- Use the Load File command from the File menu to reload the main module you unloaded, or
- Use the Set Main Module command from the Run menu to assign the role of "main module" to one of the currently loaded modules.

(QBX compile-time error)

No RESUME

The end of the program was encountered while the program was executing an error-handling routine.

Add a **RESUME** statement in the error-handling routine.

(Run-time error)

ERR code: 19

Not watchable

This error occurs when you are specifying a variable in a watch expression using the QBX Debug Menu.

Make sure the module or procedure in the active View window has access to the variable you want to watch. For example, module-level code cannot access variables that are local to a **SUB** or **FUNCTION** procedure.

(QBX run-time error)

Numeric array illegal

Numeric arrays are not allowed as arguments to **VARPTR$**. Only simple variables and string array elements are permitted.

(QBX compile-time error)

ON ERROR without /E on command line

When using the BC command, programs containing **ON** [[**LOCAL**]] **ERROR** statements must be compiled with the On Error (/E) option.

(BC compile-time error)

ON event without /v or /w on command line

When using the BC command, programs containing **ON** *event* statements (such as **ON KEY**, **ON TIMER**, or **ON COM**) must be compiled with one of these options. The /V option will cause the program to check between each program statement for occurrence of the event. The /W option will, in most cases, cause this check to be done only at each labelled statement in the program.

(BC compile-time error)

Only simple variables allowed

Names of user-defined types and arrays are not permitted as arguments in **READ** and **INPUT** statements. Array elements that are not of a user-defined type are permitted.

(Compile-time error)

Operation requires disk

You are attempting to load from, or save to, a nondisk device such as the printer or keyboard.

(QBX compile-time error)

Option unknown: *option*

You have given an illegal option.

(BC invocation error)

Out of DATA

A **READ** statement has been executed but there are no **DATA** statements with unread data remaining in the program.

(Run-time error)

ERR code: 4

Out of data space

Try modifying your data space requirements as follows:

- Use a smaller file buffer in the **OPEN** statement's **LEN** clause.
- Use the **$DYNAMIC** metacommand to create dynamic arrays. Dynamic array data can usually be much larger than static array data.

- Use the smallest data type that will accomplish your task. Use integers whenever possible.
- Try not to use many small procedures. BASIC must create several bytes of control information for each procedure.
- Use **CLEAR** to modify stack size. Use only enough stack space to accomplish your task.
- Do not use source lines longer than 255 characters. Such lines require allocation of additional text buffer space.

(Compile-time or run-time error)

Out of memory

More memory was required than is available.

General:

- Reduce the size of your DOS buffers.
- Eliminate any terminate-and-stay-resident programs.
- Eliminate some device drivers.
- In the QBX environment, unload any document, source files, or include files that are in memory but aren't needed.

Expanded memory:

- If you're using QBX and expanded memory, make sure expanded memory is available. Execute `PRINT FRE(-3)` in the Immediate window to see how much expanded memory is available. If `PRINT FRE(-3)` causes the error message `Feature unavailable` see if you are using LIM-4.0-compatible EMM memory and driver.
- Make sure you are making the best use of expanded memory. For more information, see "Memory Management for QBX" in *Getting Started.*

Extended memory:

- If you're using QBX and extended memory, make sure extended memory is available. Load QBX without the extended memory driver. Execute `PRINT FRE(-1)` in the Immediate window; note value returned. Install the extended memory driver and load QBX. Execute `PRINT FRE(-1)` in the Immediate window. If extended memory is available, the value returned should be about 60K greater than the value returned without extended memory.

DGROUP Memory:

- You may be out of space in DGROUP even if you have space left in far memory and/or extended memory. To find out how much space is left in DGROUP, execute `PRINT FRE(-1)` in the Immediate window. If the value returned is greater than 1K, you can free some space in DGROUP by reducing the number of dynamic string arrays, reducing the stack size, or putting your routines into a Quick library.

ISAM memory issues:

- If you are using ISAM, see Chapter 10, "Database Programming with ISAM" in the *Programmer's Guide*.

(BC invocation, compile-time, or run-time error)

ERR code: 7

Out of paper

The printer is out of paper or is not turned on.

(Run-time error)

ERR code: 27

Out of stack space

This error can occur when there are too many active subroutine, **FUNCTION**, or **SUB** calls, or when a recursive **FUNCTION** procedure nests too deeply.

You can change the structure of your program or use the **CLEAR** or **STACK** statement to increase the program's allotted stack space.

This error cannot be trapped.

(Run-time error)

Out of string space

There are too many string variables for the amount of memory available for strings. Strings compete with other types of program data for memory. To get more space for strings:

- In the QBX environment, unload document, source files, or include files that are not needed.
- If you are running a compiled program, use the BC command-line option /Fs (far strings) or choose the Far Strings compiler option from the QBX Run menu's Make EXE File dialog.

(Run-time error)

ERR code: 14

Overflow

The result of a calculation or data type conversion of a procedure argument is too large to be represented within the range allowed for that type of number.

This error may also occur when you come up against one of ISAM's limits. If you are creating a new index on a table:

- If there are already 40 indices existing for the table, you will have to delete an index before you can create a new one.
- The combined width of the column(s) you named for the index cannot exceed 255 bytes.

If you are opening a database table:

- If there are 32 open tables, you must close at least one before you can open another one.
- There cannot be more than 60 "long" columns already open (long columns are arrays, user-defined types, or strings with more than 255 bytes).

(Run-time error)

ERR code: 6

Overflow in numeric constant

The numeric constant is too large to be represented within the range BASIC allows for that type of number.

(Compile-time error)

Overlays incompatible with /PACKCODE

If you specify overlays which contain BASIC and non-BASIC modules, and plan to use the LINK option /PACKCODE, then the first module in every overlay must be a BASIC module. Other modules must be non-BASIC modules. Note that the LINK default is /NOPACKCODE.

(Run-time error)

Parameter type mismatch

A **SUB** or **FUNCTION** procedure parameter type does not match the **DECLARE** statement argument or the calling argument.

(Compile-time error)

Path not found

During an **OPEN**, **MKDIR**, **CHDIR**, or **RMDIR** operation, the operating system was unable to find the path specified.

The operation was not carried out.

(Run-time error)

ERR code: 76

Path/File access error

During an **OPEN**, **MKDIR**, **CHDIR**, or **RMDIR** operation, the operating system was unable to make a correct connection between the path and filename. The operation is not completed.

In the QBX environment, make sure the file specification you entered in the text box is formatted correctly. Any filename can contain a full or partial path. A full path starts with the drive name; a partial path has one or more directory names before the filename, but does not include a drive name.

This error can also occur when you try to save a file which would replace an existing read-only file.

(Compile-time or run-time error]

ERR code: 75

Permission denied

An attempt was made to write to a write-protected disk, or to access a locked file. For example, you have attempted to open a write-protected or locked file using **OPEN FOR ISAM**.

(Run-time error)

ERR code: 70

Procedure already defined in Quick library

A procedure in the Quick library has the same name as a procedure in your program.

(QBX compile-time error)

Procedure too large

The procedure has exceeded BASIC's internal size limit for a procedure. Make the procedure smaller by dividing it into several procedures.

(QBX compile-time error)

Program-memory overflow

You are attempting to compile a program whose code segment is larger than 64K. Try splitting the program into separate modules, or use the **CHAIN** statement.

(Compile-time error)

Read error on standard input

A operating-system error occurred while reading from the console or a redirected input file. The following conditions can invoke this read error:

- The compiler is invoked (using I/O redirection) with the command `BC < NEWITEM`, and the operating-system has found the file named NEWITEM, but cannot successfully open it (for example, if it is locked), this error is generated.

- The compiler is invoked with the command `BC USER,,USER;` this means that keyboard input will be the source file, and the screen will be the listing file. If the operating system cannot open the keyboard or the screen, this error is generated.

(BC invocation error)

Record/string assignment required

You did not use an appropriate data type for a string or record variable assignment in an **LSET** statement. Numeric types are not allowed.

(Compile-time error)

Record variable required

Before you can use ISAM **INSERT**, **UPDATE**, or **RETRIEVE** in statements, you must define a record variable by use of a **TYPE...END TYPE** statement.

(Compile-time error)

Redo from start

You have responded to an **INPUT** prompt with the wrong number or type of items. Retype your response in the correct form.

(Run-time prompt)

Rename across disks

An attempt was made to rename a file with a new drive designation. This is not allowed.

(Run-time prompt)

ERR code: 74

Requires DOS 2.10 or later

You are attempting to use BASIC with an incorrect version of DOS. This is a fatal, untrappable error.

(BC invocation or run-time error)

RESUME without /X on command line

When using the BC command, programs containing **RESUME**, **RESUME NEXT**, and **RESUME 0** statements must be compiled with the Resume Next (/X) option.

(BC compile-time error)

RESUME without error

A **RESUME** statement is executed, but there is no active error-handling routine.

(Run-time error)

ERR code: 20

RETURN without GOSUB

The program just executed a **RETURN** statement without executing a **GOSUB** statement.

(Run-time error)

ERR code: 3

Right parenthesis missing

BASIC expects a right (closing) parenthesis.

(BC compile-time error)

SEG or BYVAL not allowed in CALLS

BYVAL and **SEG** are permitted only in a **CALL** statement.

(BC compile-time error)

SELECT without END SELECT

The end of a **SELECT CASE** statement is missing or misspelled.

(Compile-time error)

Semicolon missing

BASIC expects a semicolon (;).

(BC compile-time error)

Separator illegal

There is an illegal delimiting character in a **PRINT USING** or **WRITE** statement. Use a semicolon or a comma as a delimiter.

(BC compile-time error)

Simple or array variable expected

The compiler expects a variable argument.

(BC compile-time error)

Skipping forward to END TYPE statement

An error in the **TYPE** statement has caused BASIC to ignore everything between the **TYPE** and **END TYPE** statements.

(BC compile-time error)

Statement cannot occur within INCLUDE file

SUB...END SUB and **FUNCTION...END FUNCTION** statement blocks are not permitted in include files.

Use the Merge command from the QBX File menu to insert the include file into the current module, or load the include file as a separate module.

If you load the include file as a separate module, some restructuring may be necessary because shared variables are shared only within the scope of the module.

(QBX compile-time error)

Statement cannot precede SUB/FUNCTION definition

The only statements allowed before a procedure definition are **REM** and **DEF***type*.

(QBX compile-time error)

Statement ignored

You are using the BC command to compile a program that contains **TRON** and **TROFF** statements without using the /D option. This error is not "fatal"; that is, the program will execute, although the results may be incorrect.

(BC compile-time warning)

Statement illegal in TYPE block

The only statements allowed between the **TYPE** and **END TYPE** statements are **REM** and the **AS** *datatype* clause.

(QBX compile-time error)

Statement unrecognizable

You have probably typed a BASIC statement incorrectly.

(BC compile-time error)

Statements/labels illegal between SELECT CASE and CASE

Statements and line labels are not permitted between **SELECT CASE** and the first **CASE** statement. Comments and statement separators are permitted.

(Compile-time error)

STOP in module-name at address *segment:offset*

A **STOP** statement was encountered in the program.

(BC run-time error)

String assignment required

The string assignment is missing from an **RSET** statement.

(BC compile-time error)

String constant required for ALIAS

The **DECLARE** statement **ALIAS** keyword requires a string-constant argument.

(BC compile-time error)

String expression required

The statement requires a string-expression argument.

(BC compile-time error)

String formula too complex

Either a string formula is too long, or an **INPUT** statement requests more than 15 string variables.

Break the formula or **INPUT** statement into smaller parts.

(Run-time error)

ERR code: 16

String space corrupt

This fatal run-time error occurs during heap compaction when an invalid string is being deleted. (Strings are kept in the heap.) The probable causes of this error are as follows:

- A string descriptor or string back pointer has been improperly modified. This may occur if you use an assembly language subroutine to modify strings.
- Out-of-range array subscripts are used and string space is inadvertently modified. The /D (Produce Debug Code) option can be used at compile time to check for array subscripts that exceed the array bounds.
- Incorrect use of the **POKE** and/or **DEF SEG** statements may modify string space improperly.
- Mismatched **COMMON** declarations may occur between two chained programs.

(BC run-time error)

String variable required

The statement requires a string-variable argument.

(BC compile-time error)

SUB or FUNCTION missing

A **DECLARE** statement has no corresponding procedure.

(BC compile-time error)

SUB/FUNCTION without END SUB/FUNCTION

The terminating statement is missing from a procedure.

(Compile-time error)

Subprogram error

This error occurs if:

- You have a label defined within a **SUB** or **FUNCTION** procedure or **DEF FN** function and you try to branch to the label from outside that procedure or function.
- You have a **RETURN** statement, with a target line number or line label, within a **SUB** or **FUNCTION** procedure.
- You use the **STATIC** statement outside a **SUB** or **FUNCTION** procedure or **DEF FN** function, or you use the **SHARED** statement outside a **SUB** or **FUNCTION** procedure.
- You place a **RUN** statement, with no filename, within a **SUB** or **FUNCTION** procedure or **DEF FN** function.

(BC compile-time error)

Subprogram not defined

A **SUB** procedure is called but never defined.

In the QBX environment, you ar attempting to call a **SUB** procedure that QBX cannot find in any loaded module or Quick library.

In the QBX environment, an ERROR 35 statement will generate this error message.

(Run-time error)

Subprograms not allowed in control statements

SUB, **FUNCTION**, or **DEF FN** definitions are not permitted inside control constructs such as **IF...THEN...ELSE** and **SELECT CASE**.

(BC compile-time error)

Subscript out of range

An array element was referred to with a subscript that was outside the dimensions of the array, or an element of an undimensioned dynamic array was accessed.

You also may get this error if the array size exceeds 64K, the array is not dynamic, and the /AH option was not used when the compiler was invoked. Reduce the size of the array, or make the array dynamic and use the /AH command-line option.

When you use the huge /AH switch, you can declare dynamic fixed-length strings and numeric arrays to a size greater than 64K. However, if you are declaring a fixed-length array and the size of its elements isn't an even power of 2, a "gap" will appear in the segment blocks. At the second occurrence of a gap, this expected error occurs. You can compensate for this by padding the type to become a size which is a power of 2.

(Run-time error)

ERR code: 9

Subscript syntax illegal

An array subscript contains a syntax error; for example, an array subscript contains both string and integer data types.

(BC compile-time error)

Symbol help not found

Symbol help is not available for your program because of errors in the program.

Use keyword help and the error message dialog boxes for help in debugging your program. Press Shift+F5 or press F8 to start running the program. This will show you what the error is.

(QBX run-time error)

Syntax error

Several conditions can cause this error:

- At compile time, the most common cause is an incorrectly typed BASIC keyword or argument.
- At run time, this error can be caused by:

  An improperly formatted **DATA** statement or when the number of key values used in an ISAM **SEEK** statement does not match the number of key values in the current index.

(Compile-time or run-time error)

ERR code: 2

Syntax error in numeric constant

A numeric constant is not properly formed.

(BC compile-time error)

Table not found

An ISAM **DELETEINDEX** statement refers to a table that does not exist.

(Run-time error)

ERR code: 82

THEN missing

BASIC expects a **THEN** keyword.

(BC compile-time error)

TO missing

BASIC expects a **TO** keyword.

(BC compile-time error)

Too many arguments in function call

Calls to **SUB** or **FUNCTION** procedures or **DEF FN** functions are limited to 60 arguments.

(BC compile-time error)

Too many dimensions

Arrays are limited to 60 dimensions.

(BC compile-time error)

Too many files

At compile time, this error occurs when include files are nested more than five levels deep.

At run time, this error may occur because:

- BASIC limits the number of disk files that can be open at one time. The limit is a function of the FILES = parameter in the CONFIG.SYS file, so increase that number and reboot.
- The operating system has a limit to the number of entries in a root directory . If your program is opening, closing, and/or saving files in the root directory, change your program so it uses a subdirectory.
- ISAM has a limit to the number of databases that can be open at one time (12 databases) and your program has exceeded this limit.

(Compile-time or run-time error)

ERR code: 67

Too many labels

The number of lines in the line list following an **ON...GOTO** or **ON...GOSUB** statement exceeds 255.

(Compile-time error)

Too many local string variables in procedure

The number of local string variables in the procedure exceeds 255. You cannot have more than 255 local string variables if you are using the /Fs option.

(Compile-time error)

Too many named COMMON blocks

The maximum number of named **COMMON** blocks permitted in a program is 126.

(BC compile-time error)

Too many TYPE definitions

The maximum number of user-defined types permitted in a program is 240.

(BC compile-time error)

Too many variables for INPUT

An **INPUT** statement is limited to 60 variables.

(BC compile-time error)

Too many variables for LINE INPUT

Only one variable is allowed in a **LINE INPUT** statement.

(BC compile-time error)

Type mismatch

Variable not of required type. This error can also occur when using ISAM in these conditions:

- During an **OPEN FOR ISAM** operation, the data type of a record element does not match the data type of a table column.
- During an ISAM **INSERT** operation, the number of elements in the record does not match the number of columns in the table, or the data type of a record element does not match the data type of a table column.
- During an ISAM **RETRIEVE** operation, the data type of a record element does not match the data type of a table column.
- In an ISAM **SEEK** statement, there is a data-type mismatch between the key-value argument and the current index.
- During an ISAM **UPDATE** operation, the record variable is not the same user-defined type as the table you are attempting to update.

(Compile-time or run-time error)

ERR code: 13

TYPE missing

The **TYPE** keyword is missing from an **END TYPE** statement.

(Compile-time error)

Type more than 65535 bytes

A user-defined type cannot exceed 64K.

(BC compile-time error)

Type not defined

Either of these conditions generates this error:

- The *usertype* argument to the **TYPE** statement is not defined.
- No user-defined type has been declared with the name used in the *tabletype* argument of an **OPEN FOR ISAM** statement.

(Compile-time error)

Type not valid for ISAM open

The **OPEN** statement in ISAM uses a user-defined type as the table type under these rules:

- The names of the fields must be less than or equal to 30 characters long.
- You cannot have an element of type **SINGLE**.

(BC compile-time error)

TYPE statement improperly nested

User-defined type definitions are not allowed in procedures.

(BC compile-time error)

TYPE without END TYPE

There is no **END TYPE** statement associated with a **TYPE** statement.

(QBX compile-time error)

Typed variable not allowed in expression

Variables that are user-defined types aren't permitted in expressions such as this, where `X` is a user-defined type:

```
CALL ALPHA((X))
```

You can pass elements of user-defined types as arguments. For example:

```
CALL Alpha((X.FirstEl))
```

(Compile-time error)

Unprintable error

An error message is not available for the error condition that exists; may be caused by **ERROR** statement that uses an error-code argument value for which BASIC has no error message string.

(Run-time error)

Valid options: [[/AH]] [[/B]] [[/C:buffersize [[{/Ea | /Es}]] [[/En]] [[/G]] [[/H]] [[/K:[[keyfile]]]] [[/L [[libraryname]]]] [[/MBF]] [[/Nofrills]] [[/NOHI]] [[[[/RUN]] programname]] [[/CMD string]]

This message appears when you invoke BASIC with an invalid option.

(QBX invocation error)

Variable-length string required

Only variable-length strings are permitted in a **FIELD** statement.

(BC compile-time error)

Variable name not unique

This error may be generated by one of the following conditions:

- You are attempting to define a variable when that variable name already exists at the same program level.
- You are attempting use a dotted variable (for example, `x.a`) in a **SHARED** statement where the name x is already being used as a record variable.
- You are attempting use a dotted variable (for example, `x.a`) in a **SUB** or **FUNCTION** procedure where x is already being used as a record variable and is not shared.

(BC compile-time error)

Variable required

At compile time, BASIC encountered an **INPUT**, **LET**, **READ**, or **SHARED** statement without a variable argument.

At run time, a **GET** or **PUT** statement didn't specify a variable when an operation was performed on a file opened in binary mode.

(Compile or run-time error)

ERR code: 40

WEND without WHILE

Each **WEND** statement must have a matching **WHILE** statement.

Also verify that other control structures within the **WHILE...WEND** structure are correctly matched. For example, an **IF** without a matching **END IF** inside the **WHILE...WEND** structure will generate this error.

In the QBX environment, an ERROR 30 statement will generate this error message.

(Compile-time error)

WHILE without WEND

Each **WHILE** statement must have a matching **WEND** statement.

In the QBX environment, an ERROR 29 statement will generate this error message.

(Compile-time error)

Wrong number of dimensions

An array reference contains the wrong number of dimensions.

(Compile-time error)

LINK Error Messages

This section lists and describes error messages generated by the Microsoft Segmented-Executable Linker, LINK.

Fatal errors cause the linker to stop execution. Fatal error messages have the following format:

location `: error L1xxx:` *messagetext*

Nonfatal errors indicate problems in the executable file. LINK produces the executable file. Nonfatal error messages have the following format:

location `: error L2xxx:` *messagetext*

Warnings indicate possible problems in the executable file. LINK produces the executable file. Warnings have the following format:

location `: warning L4xxx:` *messagetext*

In all three kinds of messages, *location* is the input file associated with the error, or LINK if there is no input file. If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown below:

```
SLIBC.LIB(_file) MAIN.OBJ(main.c) TEXT.OBJ
```

The following error messages may appear when you link object files with the Microsoft Segmented-Executable Linker, LINK.

Number	LINK Error Message
L1001	*option* **: option name ambiguous** A unique option name did not appear after the option indicator. An option is specified by an indicator (/ or –) and a name. The name can be specified by an abbreviation of the full name, but the abbreviation must be unambiguous. For example, many options begin with the letter N, so the following command causes this error: `LINK /N main;`
L1002	*option* **: unrecognized option name** An unrecognized name followed the option indicator. An option is specified by an indicator (/ or –) and a name. The name can be specified by a legal abbreviation of the full name. For example, the following command causes this error: `LINK /NODEFAULTLIBSEARCH main`
L1003	**/QUICKLIB, /EXEPACK incompatible** You cannot link with both the /QU option and the /E option.
L1004	*value* **: invalid numeric value** An incorrect value appeared for one of the linker options. For example, a character string was given for an option that requires a numeric value.
L1005	*option* **: packing limit exceeds 65536 bytes** The value supplied with the /PACKCODE or /PACKDATA option exceeds the limit of 65,536 bytes.
L1006	**/STACKSIZE : stack size exceeds 65535 bytes** The value given as a parameter to the /STACKSIZE option exceeds the maximum allowed.
L1007	**/OVERLAYINTERRUPT : interrupt number exceeds 255** A number greater than 255 was given as the /OVERLAYINTERRUPT option value. Check the *DOS Technical Reference* or other DOS technical manual for information about interrupts.
L1008	**/SEGMENTS : segment limit set too high** The /SEGMENTS option specified a limit on the number of definitions of logical segments that was impossible to satisfy.

L1009 *value* **: CPARMAXALLOC : illegal value**

The number specified in the /CPARMAXALLOC option was not in the range 1–65,535.

L1020 **no object modules specified**

No object filenames were specified to the linker.

L1021 **cannot nest response files**

A response file occurred within a response file.

L1022 **response line too long**

A line in a response file was longer than 255 characters.

L1023 **terminated by user**

You entered Ctrl+C.

L1024 **nested right parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1025 **nested left parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1026 **unmatched right parenthesis**

A left parenthesis was missing from the contents specification of an overlay on the command line.

L1027 **unmatched left parenthesis**

A right parenthesis was missing from the contents specification of an overlay on the command line.

L1030 **missing internal name**

An IMPORT statement specified an ordinal in the module-definition file without including the internal name of the routine. The name must be given if the import is by ordinal.

L1031 **module description redefined**

A DESCRIPTION statement in the module-definition file was specified more than once, a procedure that is not allowed.

L1032 **module name redefined**

The module name was specified more than once (in a NAME or LIBRARY statement), a procedure that is not allowed.

L1040 **too many exported entries**

The program exceeded the limit of 65,535 exported names.

L1041 **resident-name table overflow**

The size of the resident-name table exceeds 65,534 bytes.

An entry in the resident-name table is made for each exported routine designated RESIDENTNAME, and consists of the name plus three bytes of information. The first entry is the module name.

Reduce the number of exported routines or change some to nonresident status.

L1042 **nonresident-name table overflow**

The size of the nonresident-name table exceeds 65,534 bytes.

An entry in the nonresident-name table is made for each exported routine not designated RESIDENTNAME, and consists of the name plus three bytes of information. The first entry is the DESCRIPTION statement.

Reduce the number of exported routines or change some to resident status.

L1043 **relocation table overflow**

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Try replacing long references with short references where possible, and recreate the object module.

L1044 **imported-name table overflow**

The size of the imported-name table exceeds 65,534 bytes.

An entry in the imported-name table is made for each new name given in the IMPORTS section, including the module names, and consists of the name plus one byte.

Reduce the number of imports.

L1045 **too many TYPDEF records**

An object module contained more than 255 TYPDEF records.

These records describe communal variables. This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (TYPDEF is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1046 **too many external symbols in one module**

An object module specified more than the limit of 1,023 external symbols.

Break the module into smaller parts.

L1047 **too many group, segment, and class names in one module**

The program contained too many group, segment, and class names.

Reduce the number of groups, segments, or classes, and recreate the object file.

L1048 **too many segments in one module**

An object module had more than 255 segments.

Split the module or combine segments.

L1049 **too many segments**

The program had more than the maximum number of segments.

Use the /SEGMENTS option when linking to specify the maximum legal number of segments. The range of possible settings is 0–3,072. The default is 128.

L1050 **too many groups in one module**

LINK encountered more than 21 group definitions (GRPDEF) in a single module.

Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1051 **too many groups**

The program defined more than 20 groups, not counting DGROUP.

Reduce the number of groups.

L1052 **too many libraries**

An attempt was made to link with more than 32 libraries.

Combine libraries, or use modules that require fewer libraries.

L1053 **out of memory for symbol table**

The program had more symbolic information (such as public, external, segment, group, class, and filenames) than could fit in available memory.

Try freeing memory by linking from the operating-system command level instead of from a MAKE file or an editor. Otherwise, combine modules or segments and try to eliminate as many public symbols as possible.

L1054 **requested segment limit too high**

LINK did not have enough memory to allocate tables describing the number of segments requested. The number of segments is the default of 128 or the value specified with the /SEGMENTS option.

Try linking again by using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 **too many overlays**

The program defined more than 63 overlays.

L1057 **data record too large**

A LEDATA record (in an object module) contained more than 1,024 bytes of data. This is a translator error. (LEDATA is a DOS term that is explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module. Please report the circumstances of the error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1061 **out of memory for /INCREMENTAL**

The linker ran out of memory when trying to process the additional information required for ILINK support.

Disable incremental linking.

L1062 **too many symbols for /INCREMENTAL**

The program had more symbols than can be stored in the .SYM file.

Reduce the number of symbols or disable incremental linking.

L1063 **out of memory for CodeView information**

The linker was given too many object files with debug information, and the linker ran out of space to store it.

Reduce the number of object files that have full debug information by compiling some files with either /Zd instead of /Zi or no CodeView option at all.

L1064 **out of memory**

The linker was not able to allocate enough memory from the operating system to link the program.

Reduce the size of the program in terms of code, data, and symbols.

In OS/2, try increasing the swap space or consider splitting the program into dynamic-link libraries. Otherwise, reduce the size of the program in terms of code, data, and symbols.

L1070 *segment* **: segment size exceeds 64K**

A single segment contained more than 64K of code or data.

Try changing the memory model to use far code or data as appropriate. If the program is in C, use the /NT option or the pragma alloc_text to build smaller segments.

L1071 **segment _TEXT larger than 65520 bytes**

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.

Try compiling and linking using the medium or large model. If the program is in C, use the /NT option or the pragma alloc_text to build smaller segments.

L1072 **common area longer than 65536 bytes**

The program had more than 64K of communal variables.

This error cannot appear with object files generated by the Microsoft Macro Assembler, MASM. It occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L1073 **file-segment limit exceeded**

The number of physical or file segments exceeds the limit of 255 imposed by OS/2 protected mode and by Windows for each application or dynamic-link library.

A file segment is created for each group definition, nonpacked logical segment, and set of packed segments.

Reduce the number of segments or put more information into each segment. Make sure that the /PACKCODE and/or the /PACKDATA options are on.

L1074 *group* **: group larger than 64K bytes**

The given group exceeds the limit of 65,536 bytes.

Reduce the size of the group, or remove any unneeded segments from the group. Refer to the map file for a listing of segments.

L1075 **entry table larger than 65535 bytes**

The entry table exceeds the limit of 65,535 bytes.

There is an entry in this table for each exported routine for each address that is the target of a far relocation, and for one of the following conditions when true:

- The target segment is designated IOPL (specific to OS/2).
- PROTMODE is not enabled and the target segment is designated MOVABLE (specific to Windows).

Declare PROTMODE if applicable, or reduce the number of exported routines, or make some segments FIXED or NOIOPL if possible.

L1078 **file-segment alignment too small**

The segment-alignment size given with the /ALIGN:*number* option was too small. Try increasing the number.

L1080 **cannot open list file**

The disk or the root directory was full.

Delete or move files to make space.

L1081 **out of space for run file**

The disk on which the .EXE file was being written was full.

Free more space on the disk and restart the linker.

L1082 *filename* **: stub file not found**

The linker could not open the file given in the STUB statement in the module-definition file.

The file must be in the current directory or in a directory specified by the PATH environment variable.

L1083 **cannot open run file**

The disk or the root directory was full.

Delete or move files to make space.

L1084 **cannot create temporary file**

The disk or root directory was full.

Free more space in the directory and restart the linker.

L1085 **cannot open temporary file**

The disk or the root directory was full.

Delete or move files to make space.

L1086 **scratch file missing**

An internal error has occurred.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1087 **unexpected end-of-file on scratch file**

The disk with the temporary linker-output file was removed.

L1088 **out of space for list file**

The disk where the listing file was being written is full.

Free more space on the disk and restart the linker.

L1089 *filename* **: cannot open response file**

LINK could not find the specified response file.

This usually indicates a typing error.

L1090 **cannot reopen list file**

The original disk was not replaced at the prompt.

Restart the linker.

L1091 **unexpected end-of-file on library**

The disk containing the library was probably removed.

Replace the disk containing the library and run the linker again.

L1092 **cannot open module-definitions file**

The linker could not open the module-definition file specified on the command line or in the response file.

L1093 *filename* **: object not found**

One of the object files specified in the linker input was not found.

Restart the linker and specify the object file.

L1094 *filename* **: cannot open file for writing**

The linker was unable to open the file with write permission.

Check file permissions.

L1095 *filename* **: out of space on file**

The linker ran out of disk space for the specified output file.

Delete or move files to make space.

L1100 **stub .EXE file invalid**

The file specified in the STUB statement is not a valid real-mode executable file.

L1101 **invalid object module**

One of the object modules was invalid.

Check that the correct version of the linker is being used.

If the error persists after recompiling, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1102 **unexpected end-of-file**

An invalid format for a library was encountered.

L1103 **attempt to access data outside segment bounds**

A data record in an object module specified data extending beyond the end of a segment. This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1104 *filename* **: not valid library**

The specified file was not a valid library file. This error causes LINK to abort.

L1105 **invalid object due to aborted incremental compile**

Delete the object file, recompile the program, and relink.

L1113 **unresolved COMDEF; internal error**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1114 **file not suitable for /EXEPACK; relink without**

For the linked program, the size of the packed load image plus packing overhead was larger than that of the unpacked load image.

Relink without the /EXEPACK option.

L1115 *option* **: option incompatible with overlays**

The given option is not compatible with overlays. Remove the option or else do not use overlaid modules.

L1123 *segment* **: segment defined for both 16- and 32-bit.**

Define the segment as either 16-bit or 32-bit.

L1126 **conflicting IOPL-parameter-words value**

An exported name was specified in the module-definition file with an IOPL-parameter-words value, and the same name was specified as an export by the Microsoft C export pragma with a different parameter-words value.

L1127 **far segment references not allowed with /TINY**

The /TINY option (causing the linker to produce a .COM file) was used with modules that have a far segment reference.

Far segment references are not compatible with the .COM file format. High-level language modules cause this error (unless the language supports tiny memory model). Assembly code referencing a segment address also produces this error. For example:

```
movax, seg mydata
```

L2000 **imported starting address**

The program starting address as specified in the END statement in a MASM file is an imported routine. This is not supported by OS/2 or Windows.

L2002 **fixup overflow at *number* in segment *segment***

This error message will be followed by one of the following:

- `Target external` *symbol*
- `frm seg` *name1,* `tgt seg` *name2,* `tgt offset` *number*

A fixup overflow is an attempted reference to code or data that is impossible because the source location (where the reference is made "from") and the target address (where the reference is made "to") are too far apart. Usually the problem is corrected by examining the source location.

Revise the source file and recreate the object file. (For information about frame and target segments, see the *Microsoft MS-DOS Programmer's Reference*.)

L2003 **intersegment self-relative fixup at *number* in segment *segment***

The program issued a near call or jump to a label in a different segment.

This error most often occurs when you specifically declare an external procedure to be near and it should be declared as far.

L2005 **fixup type unsupported at *number* in segment *segment***

A fixup type occurred that is not supported by the Microsoft linker. This is probably a compiler error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2010 **too many fixups in LIDATA record**

The number of far relocations (pointer- or base-type) in a LIDATA record exceeds the limit imposed by the linker.

This is typically produced by the DUP statement in an .ASM file. The limit is dynamic: a 1,024-byte buffer is shared by relocations and the contents of the LIDATA record; there are eight bytes per relocation.

Reduce the number of far relocations in the DUP statement.

L2011 *identifier* **: NEAR/HUGE conflict**

Conflicting NEAR and HUGE attributes were given for a communal variable.

This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L2012 *arrayname* **: array-element size mismatch**

A far communal array was declared with two or more different array-element sizes (for instance, an array was declared once as an array of characters and once as an array of real numbers).

This error occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.

L2013 **LIDATA record too large**

An LIDATA record contained more than 512 bytes. This is probably a compiler error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2022 *routine* **(alias** *internalname***) : export undefined**

The internal name of the given exported routine is undefined.

L2023 *routine* **(alias** *internalname***) : export imported**

The internal name of the given exported routine conflicts with the internal name of a previously imported routine.

Make sure that the set of imported and exported names do not overlap.

L2024 *symbol* **: special symbol already defined**

The program defined a symbol name already used by the linker for one of its own low-level symbols. For example, the linker generates special symbols used in overlay support and other operations.

Choose another name for the symbol in order to avoid conflict.

L2025 *symbol* **: symbol defined more than once**

The same symbol has been found in two different object files.

L2026 **entry ordinal** *number*, **name** *name* **: multiple definitions for same ordinal**

The given exported name with the given ordinal number conflicted with a different exported name previously assigned to the same ordinal. Only one name can be associated with a particular ordinal.

L2027 *name* **: ordinal too large for export**

The given exported name was assigned an ordinal that exceeded the limit of 65,535.

L2028 **automatic data segment plus heap exceed 64K**

The total size of data declared in DGROUP, plus the value given in HEAPSIZE in the module-definition file, plus the stack size given by the /STACKSIZE option or STACKSIZE module-definition file statement, exceeds 64K.

Reduce near-data allocation, HEAPSIZE, or the stack.

L2029 *symbol* **: unresolved external**

The name of the unresolved external symbol is given first, then a list of object modules that contain references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 **starting address not code (use class 'CODE')**

The program starting address, as specified in the END statement of an .ASM file, should be in a code segment. Code segments are recognized if their class name ends in 'CODE'. This is an error in OS/2 protected mode.

The error message may be disabled by including the REALMODE statement in the module-definition file.

L2041 **stack plus data exceed 64K**

If the total of near data and requested stack size exceeds 64K, the program will not run correctly. The linker checks for this condition only when /DOSSEG is enabled, which is the case in the library startup module. This is a fatal error.

Reduce the stack size.

L2043 **Quick Library support module missing**

The required module QUICKLIB.OBJ was missing.

The module QUICKLIB.OBJ must be linked when creating a Quick library.

L2044 *symbol* **: symbol multiply defined, use /NOE**

The linker found what it interprets as a public-symbol redefinition, probably because you have redefined a symbol defined in a library.

Relink with the /NOEXTDICTIONARY (NOE) option. If error L2025 results for the same symbol, then you have a genuine symbol-redefinition error.

L2045 *segment* **: segment with > 1 class name not allowed with /INC**

Your program defined a segment more than once, giving the segment different class names. Different class names for the same segment are not allowed when you link with the /INCREMENTAL option. Normally, this error should never appear unless you are programming with MASM. For example, if you give the following two MASM statements, then the statements have the effect of defining two distinct segments with the same name but different classes:

```
_BSS segment 'BSS' _BSS segment 'DATA'
```

This situation is incompatible with the /INCREMENTAL option.

L2047 **IOPL attribute conflict - segment:** *segment* **in group:** *group*

The specified segment is a member of the specified group, but has a different IOPL attribute from other segments in the group.

L2048 **Microsoft Overlay Manager module not found**

Overlays were designated, but the Microsoft Overlay Manager module was not found. This module is defined in the default library.

L2049 **no segments defined**

No code or initialized data was defined in the program. The resulting executable file is not likely to be valid.

L2050 **16/32 bit attribute conflict - segment:** *segment* **in group:** *group*

You cannot group 16-bit segments with 32-bit segments.

L2051 **start address not equal to 0x100 for /TINY**

The program starting address as specified in the .COM file must have a starting value equal to 100 hex (0x100 or 0x0). Any other value is illegal.

Put assembly source code ORG 100h in front of the code segment.

L2052 *symbol*: **unresolved external - possible calling convention mismatch**

A symbol was declared to be external in one or more modules, but LINK could not find it defined as public in any module or library.

This error occurs when a prototype for an externally-defined function is omitted from a C program that is compiled with the fastcall option (/Ox). The calling convention for fastcall does not match the assumptions that are made when a prototype is not included for an external function.

Either include a prototype for the function, or compile without the /Ox option.

The name of the unresolved external symbol is given first, then a list of object modules that contain references to this symbol. This message and the list are also written to the map file, if one exists.

L4000 **seg disp. included near** *offset* **in segment** *segment*

This is the warning generated by the /WARNFIXUP option.

L4001 **frame-relative fixup, frame ignored near** *offset* **in segment** *segment*

A reference is made relative to a segment that is different from the target segment of the reference.

For example, if `_proc1` is defined in segment `_TEXT`, the instruction call `DGROUP:_proc1` produces this warning. The frame DGROUP is ignored, so the linker treats the call as if it were call `_TEXT:_proc1`.

L4002 **frame-relative absolute fixup near** *offset* **in segment** *segment*

A reference is made relative to a segment that is different from the target segment of the reference, and both segments are absolute (defined with AT). The linker treats the executable file as if the file were to run in real mode only.

L4003 **intersegment self-relative fixup at** *offset* **in segment** *segment* **pos:** *offset* **target external** *name*

The linker found an intersegment self-relative fixup. This error may be caused by compiling a small-model program with the /NT option.

L4004 **possible fixup overflow at** *offset* **in segment** *segment*

This warning is issued for DOS programs when a near call/jump is made to another segment which is not a member of the same group as the segment from which the call/jump was made. This call/jump can cause an incorrect real-mode address calculation when the distance between paragraph address (frame number) of the segment group and the target segment is greater than 64K even when the distance between the segment where the call/jump was actually made and the target segment is less than 64K.

L4010 **invalid alignment specification**

The number specified in the /ALIGNMENT option must be a power of 2 in the range 2–32,768.

L4011 **PACKCODE value exceeding 65500 unreliable**

The packing limit specified with the /PACKCODE option was between 65,500 and 65,536. Code segments with a size in this range are unreliable on some steppings of the 80286 processor.

L4012 **load-high disables /EXEPACK**

The /HIGH and /EXEPACK options cannot be used at the same time.

L4013 *option* **: option ignored for segmented-executable file**

The use of overlays and options /CPARMAXALLOC, /DSALLOCATE, and /NOGROUPASSOCIATION is not allowed with either OS/2 protected-mode or Windows executable files.

L4014 *option* **: option ignored for realmode executable file**

The /ALIGNMENT option is invalid for real-mode executable files.

L4015 **/CODEVIEW disables /DSALLOCATE**

The /CODEVIEW and /DSALLOCATE options cannot be used at the same time.

L4016 **/CODEVIEW disables /EXEPACK**

The /CODEVIEW and /EXEPACK options cannot be used at the same time.

L4017 *option* **: unrecognized option name; option ignored**

An unrecognized character followed the option indicator (/), as shown below:

```
LINK /ABCDEF main;
```

L4018 **missing or bad application type; option** ***option*** **ignored**

The /PMTYPE option accepts only:

- Presentation Manager applications that use the Presentation Manager API and are executed in the Presentation Manager environment.
- Presentation-Manager-compatible applications that run in the Presentation Manager environment from a VIO window.
- Presentation-Manager-compatible or -incompatible applications that run in a separate screen group.

L4019 **/ TINY disables / INCREMENTAL**

A .COM file always requires a full link and cannot be incrementally linked. / TINY and / INCREMENTAL are incompatible, and when they are used together, the linker will ignore / INCREMENTAL.

L4020 *segment* **: code-segment size exceeds 65500**

Code segments of 65,501–65,536 bytes in length may be unreliable on the Intel 80286 processor.

L4021 **no stack segment**

The program did not contain a stack segment defined with STACK combine type.

Normally, every program should have a stack segment with the combine type specified as STACK. You may ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type. Linking with versions of LINK earlier than Version 2.40 might cause this message since these linkers search libraries only once.

L4022 *group1*, *group2* **: groups overlap**

The named groups overlap. Since a group is assigned to a physical segment, groups cannot overlap with either OS/2 protected-mode or Windows executable files.

Reorganize segments and group definitions so the groups do not overlap. Refer to the map file.

L4023 *routine* (*internalname*) **: export internal name conflict**

The internal name of the given exported routine conflicted with the internal name of a previous import definition or export definition.

L4024 *name* **: multiple definitions for export name**

The given name was exported more than once, an action that is not allowed.

L4025 *modulename.importname*(*internalname*) **: import internal name conflict**

The internal name of the given imported routine (import is either a name or a number) conflicted with the internal name of a previous export or import.

L4026 *modulename.importname*(*internalname*) **: self-imported**

The given imported routine was imported from the module being linked. This is not supported on some systems.

L4027 *name* **: multiple definitions for import internal-name**

The given internal name was imported more than once. Previous import definitions are ignored.

L4028 *segment* **: segment already defined**

The given segment was defined more than once in the SEGMENTS statement of the module-definition file.

L4029 *segment* **: DGROUP segment converted to type data**

The given logical segment in the group DGROUP was defined as a code segment.

DGROUP cannot contain code segments because the linker always considers DGROUP to be a data segment. The name DGROUP is predefined as the automatic data segment.

The linker converts the named segment to type DATA.

L4030 *segment* **: segment attributes changed to conform with automatic data segment**

The given logical segment in the group DGROUP was given sharing attributes (SHARED/NONSHARED) that differed from the automatic data attributes as declared by the DATA instance specification (SINGLE/MULTIPLE). The attributes are converted to conform to those of DGROUP.

The name DGROUP is predefined as the automatic data segment. DGROUP cannot contain code segments because the linker always considers DGROUP to be a data segment.

L4031 *segment* **: segment declared in more than one group**

A segment was declared to be a member of two different groups.

Correct the source file and recreate the object files.

L4032 *segment* **: code-group size exceeds 65500 bytes**

The given code group has a size between 65,500 and 65,536 bytes, a size that is unreliable on some steppings of the 80286 processor.

L4034 **more than 239 overlay segments; extra put in root**

The LINK command line or response file designated too many segments to go into overlays.

The limit on the number of segments that can go into overlays is 239. Segments starting with the 240th segment are assigned to the permanently resident portion of the program (the root).

L4036 **no automatic data segment**

The application did not define a group named DGROUP. DGROUP has special meaning to the linker, which uses it to identify the automatic or default data segment used by the operating system. Most OS/2 protected-mode and Windows applications require DGROUP. This warning will not be issued if DATA NONE is declared or if the executable file is a dynamic-link library.

L4038 **program has no starting address**

Your OS/2 or Windows application had no starting address, which usually causes the program to fail. Higher-level languages automatically specify a starting address. If you are writing an assembly-language program, specify a starting address with the END statement.

Real-mode programs and dynamic-link libraries should never receive this message, regardless whether or not they have starting addresses.

L4040 **stack size ignored for / TINY**

The linker will ignore stack size if /TINY is given and if the stack segment has been defined in front of the code segment.

L4042 **cannot open old version**

The file specified in the OLD statement in the module-definition file could not be opened.

L4043 **old version not segmented-executable format**

The file specified in the OLD statement in the module-definition file was not a valid OS/2 protected-mode or Windows executable file.

L4045 **name of output file is** *filename*

The linker had to change the name of the output file to the given filename.

If the output file is specified without an extension, the linker assumes the default extension .EXE. Creating a Quick library, .DLL file, or .COM file forces the linker to use an extension other than .EXE as follows:

Output file specification	Extension
/TINY option	.COM
/QUICKLIBRARY option	.QLB
LIBRARY statement in .DEF file	.DLL

L4047 **Multiple code segments in module of overlaid program incompatible with /CODEVIEW**

When debugging with CodeView, if there are multiple code segments defined in one module (.OBJ file) by use of the compiler pragma alloc_text( ), and the program is built as an overlaid program, you can access the symbolic information for only the first code segment in the overlay. Symbolic information for the rest of the code segments in the overlay is not accessible.

L4050 **too many public symbols for sorting**

The linker uses the stack and all available memory in the near heap to sort public symbols for the /MAP option. If the number of public symbols exceeds the space available for them, this warning is issued and the symbols are not sorted in the map file but listed in an arbitrary order.

Reduce the number of symbols.

L4051 *filename* **: cannot find library**

The linker could not find the specified file.

Enter a new file name, a new path specification, or both.

L4053 **VM.TMP : illegal file name; ignored**

VM.TMP appeared as an object filename.

Rename the file and rerun the linker.

L4054 *filename* **: cannot find file**

The linker could not find the specified file.

Enter a new filename, a new path specification, or both.

LIB Error Messages

Error messages generated by the Microsoft Library Manager, LIB, have one of the following formats:

```
{filename LIB} : fatal error U1xxx: messagetext
{filename LIB} : nonfatal error U2xxx: messagetext
{filename LIB} : warning U4xxx: messagetext
```

The message begins with the input filename (*filename*), if one exists, or with the name of the utility. If possible, LIB prints a warning and continues operation. In some cases errors are fatal, and LIB terminates processing.

LIB may display the following error messages.

Number	LIB Error Message
U1150	**page size too small** The page size of an input library was too small, indicating an invalid input .LIB file.
U1151	**syntax error : illegal file specification** A command operator such as a hyphen (-) was given without a following module name.
U1152	**syntax error : option name missing** A forward slash (/) was given without an option after it.
U1153	**syntax error : option value missing** The /PAGESIZE option was given without a value following it.
U1154	**option unknown** An unknown option was given. LIB recognizes the /HELP, /?, /IGNORECASE, /NOIGNORECASE, /NOEXTDICTIONARY, /NOLOGO, and /PAGESIZE, and options.

U1155 **syntax error : illegal input**

The given command did not follow correct LIB syntax.

U1156 **syntax error**

The given command did not follow correct LIB syntax.

U1157 **comma or new line missing**

A comma or carriage return was expected in the command line but did not appear. Alternatively, this message may indicate an incorrectly placed comma, as in the following line:

```
LIB math.lib,-mod1+mod2;
```

The line should have been entered as follows:

```
LIB math.lib -mod1+mod2;
```

U1158 **terminator missing**

Either the response to the Output library prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 **cannot rename old library**

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read-only protection.

Change the protection on the old .BAK version.

U1162 **cannot reopen library**

The old library could not be reopened after it was renamed to have a .BAK extension.

U1163 **error writing to cross-reference file**

The disk or root directory was full.

Delete or move files to make space.

U1164 **name length exceeds 255 characters**

LIB has a limit of 255 characters. Reduce the number of characters in the name.

U1170 **too many symbols**

More than 4,609 symbols appeared in the library file.

U1171 **insufficient memory**

LIB did not have enough memory to run.

Remove any shells or resident programs and try again, or add more memory.

U1172 **no more virtual memory**

The current library exceeds the one-megabyte limit imposed by LIB.

Try using the /NOEXTDICTIONARY option or reduce the number of object modules.

U1173 **internal failure**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1174 *mark*: **not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1175 *free*: **not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1180 **write to extract file failed**

The disk or root directory was full.

Delete or move files to make space.

U1181 **write to library file failed**

The disk or root directory was full.

Delete or move files to make space.

U1182 *filename* **: cannot create extract file**

The disk or root directory was full, or the specified extract file already existed with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 **cannot open response file**

The response file was not found.

U1184 **unexpected end-of-file on command input**

An end-of-file character was received prematurely in response to a prompt.

U1185 **cannot create new library**

The disk or root directory was full, or the library file already existed with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 **error writing to new library**

The disk or root directory was full.

Delete or move files to make space.

U1187 **cannot open VM.TMP**

The disk or root directory was full.

Delete or move files to make space.

U1188 **cannot write to VM**

The library manager cannot write to the virtual memory.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1189 **cannot read from VM**

The library manager cannot read the virtual memory.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1190 **interrupted by user**

LIB was interrupted during its operation, with Ctrl+C or Ctrl+Break.

U1200 *filename* **: invalid library header**

The input library file had an invalid format. Either it was not a library file or it had been corrupted.

U1203 *filename* **: invalid object module near location**

The module specified by *filename* was not a valid object module.

U2152 *filename* **: cannot create listing**

The directory or disk was full, or the cross-reference-listing file already existed with read-only protection.

Make space on the disk or change the protection of the cross-reference-listing file.

U2155 *module* **: module not in library; ignored**

The specified module was not found in the input library.

U2157 *filename* **: cannot access file**

LIB was unable to open the specified file.

U2158 *library* **: invalid library header; file ignored**

The input library had an incorrect format.

U2159 *filename* **: invalid format number (***number***); file ignored**

The signature byte or word number of the given file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or XENIX archive.

U4150 *module* **: module redefinition ignored**

A module was specified to be added to a library but a module with the same name was already in the library, or a module with the same name was found more than once in the library.

U4151 *symbol* **: symbol defined in module** *module-name***, redefinition ignored**

The specified symbol was defined in more than one module.

U4153 **pagesize:** *value* **: page size invalid; ignored**

The value specified in the /PAGESIZE option was less than 16.

U4155 *module* **: module not in library**

A module specified to be replaced does not already exist in the library. LIB adds the module anyway.

U4156 *libraryname* **: output-library specification ignored**

An output library was specified in addition to a new library name. For example, specifying the following command line, where `new.lib` does not already exist, causes this error:

```
LIB new.lib+one.obj,new.lst,new.lib
```

U4157 **insufficient memory, extended dictionary not created**

Insufficient memory prevented LIB from creating an extended dictionary. The library is still valid, but the linker will not be able to take advantage of the extended dictionary to speed linking.

U4158 **internal error, extended dictionary not created**

An internal error prevented LIB from creating an extended dictionary. The library is still valid, but the linker cannot take advantage of the extended dictionary to speed linking.

HIMEM.SYS Error Messages

An Extended Memory Manager is already installed. XMS Driver not installed.

You have tried to install HIMEM.SYS after it has already been installed on your system. HIMEM.SYS can be installed only once.

HIMEM.SYS requires an 80x86-based machine. XMS Driver not installed.

You have tried to install HIMEM.SYS on a machine without an 80286 or later microprocessor. HIMEM.SYS can be installed only on a computer system which has an 80286 or later microprocessor.

HIMEM.SYS requires DOS 3.00 or higher. XMS Driver not installed.

You have tried to install HIMEM.SYS on a machine using a version of MS-DOS lower than 3.0. You can install HIMEM.SYS only on systems using MS-DOS version 3.0 or higher.

No available extended memory was found.

You have tried to install HIMEM.SYS on a computer without any extended memory. HIMEM.SYS can be installed only on a computer with extended memory.

Unrecognized A20 hardware.

HIMEM.SYS cannot recognize the A20 hardware of your system. The A20 address line on your microprocessor allows access to a 64K segment of memory, known as the High Memory Area. If this occurs, it is probably because the system is not one supported by this release of HIMEM.SYS. Contact your computer manufacturer or dealer to see if an extended memory driver exists for your machine.

WARNING: The A20 Line was already enabled.

During the installation process, HIMEM.SYS detected the A20 hardware was already enabled. The A20 address line on your microprocessor allows access to a 64K segment of memory, known as the High Memory Area. HIMEM.SYS will remain installed and attempt to work properly, however it will not disable the A20 line.

WARNING: The High Memory Area is unavailable.

HIMEM.SYS cannot find enough memory to use the High Memory Area. HIMEM.SYS will not be able to process any requests for the High Memory Area. However, HIMEM.SYS will remain installed to process any requests for the Extended Memory Data Blocks.

RAMDRIVE.SYS Error Messages

RAMDrive: Computer must be PC-AT, or PC-AT compatible

The /E switch was specified on a computer other than an IBM PC AT (or 100% compatible), AT&T 6300 PLUS, or a 386 with extended memory. As a result, RAMDRIVE.SYS was not installed.

Remove the /E parameter and try to install RAMDrive in system memory.

RAMDrive: Expanded Memory Manager not present

You included the /A switch in the RAMDrive command line but RAMDrive could not find the expanded memory manager. Your system boot disk did not install the expanded memory manager. Your CONFIG.SYS file did not contain the appropriate information. As a result, RAMDRIVE.SYS was not installed.

Consult the documentation for your expanded memory hardware for correct installation instructions.

RAMDrive: Expanded Memory Status shows error

While trying to set up RAMDrive in expanded memory, DOS detected an error. As a result, RAMDRIVE.SYS was not installed.

Run your expanded memory diagnostics to check your expanded memory. Take the appropriate corrective action as instructed in your expanded memory manual.

RAMDrive: Incorrect DOS version

RAMDrive requires DOS version 2.*X* or later. As a result, RAMDRIVE.SYS was not installed.

Upgrade to DOS 2.0 or later.

RAMDrive: Insufficient memory

Your computer has some memory available, but not enough to set up a RAMDrive. As a result, RAMDRIVE.SYS was not installed.

Free some extended memory or obtain more memory.

RAMDrive: Invalid parameter

The parameters you specified in your CONFIG.SYS entry for RAMDRIVE.SYS are incorrect. As a result, RAMDRIVE.SYS was not installed.

Check to see if you specified too many parameters, if one of your numeric parameters is not valid, if you specified conflicting switches (i.e. only one of /A or /E may be specified), or if you specified too many switches. Change the RAMDRIVE.SYS command line in your CONFIG.SYS file to conform to the usage described above.

RAMDrive: I/O error accessing drive memory

During the set up of the RAMDrive, an error was detected in the memory being accessed for RAMDrive. As a result, RAMDRIVE.SYS was not installed.

Run the memory test for the memory on which you were attempting to install a RAMDrive.

RAMDrive: No extended memory available

Your computer has no memory available for RAMDrives. As a result, RAMDRIVE.SYS was not installed.

Free some extended memory or obtain more memory.

SMARTDRV.SYS Error Messages

SMARTDrive : Expanded Memory Manager not present

You included the /A switch in the SMARTDrive command line but SMARTDrive could not find the expanded memory manager. Your system boot disk did not install the expanded memory manager. Your CONFIG.SYS file did not contain the appropriate information.

Consult the documentation for your expanded memory hardware for correct installation instructions.

SMARTDrive : Expanded Memory Status shows error

While trying to set up SMARTDrive in expanded memory, DOS detected an error. DOS will not install the SMARTDrive program.

Run your expanded memory diagnostics to check your expanded memory. Take the appropriate corrective action as instructed in your expanded memory manual.

SMARTDrive : Incorrect DOS version

SMARTDrive runs only on DOS version 2.0 or later. DOS will not install the SMARTDrive program.

You need to switch to a 2.0 or later version of DOS so that you can run SMARTDrive.

SMARTDrive : Insufficient memory

Your system has insufficient memory available for SMARTDrive. DOS will not install the SMARTDrive program.

If you want to use the SMARTDrive program, you must add memory to your system.

SMARTDrive : Invalid parameter

The command line contains too many parts, such as more than one number or more than one path.

The size number is out of the range of permitted numbers.For example, you may have the SMARTDrive size set for 8K, which is too small. DOS will not install the SMARTDrive program.

Edit your CONFIG.SYS file and change the incorrect SMARTDrive line.

SMARTDrive : I/O error accessing cache memory

DOS detected an error while trying to set up SMARTDrive. DOS will not install the SMARTDrive program.

Run memory tests to check the memory where SMARTDrive is set up.

SMARTDrive : No extended memory available

Your system has no memory available for SMARTDrive. DOS will not install the SMARTDrive program.

If you want to use the SMARTDrive program, you must add memory to your system.

SMARTDrive : No hard drives on system

Your system has no hard disk. DOS will not install the SMARTDrive program. SMARTDrive only works with hard disks.

If you want to use the SMARTDrive program, you must add a hard disk to your system.

SMARTDrive : Too many bytes per track on hard drive.

Your system has a hard disk that SMARTDrive does not understand. DOS will not install the SMARTDrive program.

You cannot use SMARTDrive on your current system.

NMAKE Error Messages

Error messages from the NMAKE utility have one of the following formats:

{*filename* | `NMAKE`} `: fatal error U1xxx:` *messagetext*
{*filename* | `NMAKE`} `: warning U4xxx:` *messagetext*

The message begins with the input filename (*filename*) and line number, if one exists, or with the name of the utility.

NMAKE generates the following error messages.

Number	NMAKE Error Message

U1000 **syntax error : ')' missing in macro invocation**

A left parenthesis, (, appeared without a matching right parenthesis,), in a macro invocation. The correct form is $(*name*), or $*n* for one-character names.

U1001 **syntax error : illegal character '*character*' in macro**

A nonalphanumeric character other than an underscore (_) appeared in a macro.

U1002 **syntax error : bad macro invocation '$'**

A single dollar sign ($) appeared without a macro name associated with it.

The correct form is $(*name*). To use a dollar sign in the file, type it twice ($$) or precede it with a caret (^).

U1003 **syntax error : '=' missing in macro**

The equal sign (=) was missing in a macro definition. The correct form is '*name* = *value*'.

U1004 **syntax error : macro name missing**

A macro invocation appeared without a name.

The correct form is $(*name*).

U1005 **syntax error : text must follow ':' in macro**

A string substitution was specified for a macro, but the string to be changed in the macro was not specified.

U1006 **extmake syntax in '*string*' incorrect**

The part of the string shown contains an extmake syntax error.

U1007 **extmake syntax usage error, no dependent**

No dependent was given. In extmake syntax, the target under consideration must have either an implicit dependent or an explicit dependent.

U1008 **syntax error : only (no)keep allowed here**

Something other than KEEP or NOKEEP appeared at the end of the syntax for creating an inline file. The syntax for generating an inline file allows an action to be specified after the second pair of angle brackets. Valid actions are KEEP and NOKEEP. Other actions are errors.The KEEP option specifies that NMAKE should leave the inline file on disk. The NOKEEP option causes NMAKE to delete the file before exiting. The default is NOKEEP.

U1009 **expanded command line '*commandline*' too long**

After macro expansion, the command line shown exceeded the length limit for command lines

for the operating system. DOS permits up to 128 characters on a command line. If the command is a LINK command line, use a response file.

U1010 **cannot open file** **'*filename*'**

The given file could not be opened, either because the disk was full or because the file has been set to be read-only.

U1016 **syntax error : missing closing double quotation mark**

An opening double quotation mark (") appeared without a closing double quotation mark.

U1017 **unknown directive** **'\!*directive*'**

The directive specified is not one of the recognized directives.

U1018 **directive and/or expression part missing**

The directive was incompletely specified. The expression part is required.

U1019 **too many nested if blocks**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1020 **end-of-file found before next directive**

A directive, such as !ENDIF, was missing.

U1021 **syntax error : else unexpected**

An !ELSE directive was found that was not preceded by !IF, !IFDEF, or !IFNDEF; or was placed in a syntactically incorrect place.

U1022 **missing terminating character for string/program invocation :** **'*char*'**

The closing double quotation mark (") in a string comparison in a directive was missing. Or, the closing bracket (]) in a program invocation in a directive was missing.

U1023 **syntax error present in expression**

An expression is invalid. Check the allowed operators and operator precedence.

U1024 **illegal argument to !CMDSWITCHES**

An unrecognized command switch was specified.

U1031 **file name missing (or macro is null)**

An include directive was found, but the name of the file to be included was missing; or the macro expanded to nothing.

U1033 **syntax error : '*string*' unexpected**

The specified string is not part of the valid syntax for a makefile.

U1034 **syntax error : separator missing**

The colon (:) that separates target(s) and dependent(s) is missing.

U1035 **syntax error : expected ':' or '=' separator**

Either a colon (:), implying a dependency line, or an equal sign (=), implying a macro definition, was expected.

U1036 **syntax error : too many names to left of '='**

Only one string is allowed to the left of a macro definition.

U1037 **syntax error : target name missing**

A colon (:) was found before a target name was found. At least one target is required.

U1038 **internal error : lexer**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1039 **internal error : parser**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1040 **internal error : macroexpansion**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1041 **internal error : target building**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1042 **internal error : expression stack overflow**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1043 **internal error : temp file limit exceeded**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1044 **internal error : too many levels of recursion building a target**

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1045 *'internal error message'*

Note the circumstances of the error and notify Microsoft by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1050 *'user-specified text'*

The message specified with the !ERROR directive is displayed.

U1051 **out of memory**

The program ran out of space in the far heap. Split the description file into smaller and simpler pieces.

U1052 **file '*filename*' not found**

The file was not found. The filename may not be properly specified in the makefile.

U1053 **file '*filename*' unreadable**

The file cannot be read. The following are possible causes of this error:

- The file does not have appropriate attributes for reading.
- A bad area exists on disk.
- A bad file-allocation table exists.
- The file is locked.

U1054 **cannot create in-line file '*filename*'**

NMAKE failed in its attempt to create the file given by *filename*. The following are possible causes of this error:

- The file already exists with a read-only attribute.
- There is insufficient disk space to create the file.

U1055 **out of environment space**

The environment space limit was reached. Restart the program with a larger environment space or with fewer environment variables.

U1056 **cannot find command processor**

The command processor was not found. NMAKE uses COMMAND.COM or CMD.EXE as a command processor to execute commands. It looks for the command processor first by the full path given by the COMSPEC environment variable. If COMSPEC does not exist, NMAKE searches the directories specified by the PATH environment variable.

U1057 **unlink of file '*filename*' failed**

NMAKE failed to delete the temporary inline file.

U1058 **terminated by user**

Execution of NMAKE was aborted by Ctrl+C or Ctrl+Break.

U1060 **unable to close file : '*filename*'**

NMAKE encountered an error while closing a file.

One of the following may have occurred:

- The file is a read-only file.
- There is a locking or sharing violation.
- The disk is full.

U1061 **/F option requires a file name**

The /F command-line option requires the name of the description file to be specified. To use standard input, specify '-' as the description filename.

U1062 **missing file name with /X option**

The /X command-line option requires the name of the file to which diagnostic error output should be redirected. To use standard input, specify '-' as the output filename.

U1063 **missing macro name before '='**

NMAKE detected an equal sign (=) without a preceding name. This error can occur in a recursive call when the macro corresponding to the macro name expands to nothing.

U1064 **MAKEFILE not found and no target specified**

No description file was found, and no target was specified. A description file can be specified either with the /F option or through the default file MAKEFILE. Note that NMAKE can create a target using an inference rule even if no description file is specified.

U1065 **invalid option '*option*'**

The option specified is not a valid option for NMAKE.

U1070 **cycle in macro definition '*macroname*'**

A circular definition was detected in the macro definition specified. Circular definitions are invalid.

U1071 **cycle in dependency tree for target '*targetname*'**

A circular dependency was detected in the dependency tree for the specified target. Circular dependencies are invalid.

U1072 **cycle in include files : '*filename*'**

A circular inclusion was detected in the include file specified. The named file includes a file which eventually includes the named file.

U1073 **don't know how to make '*targetname*'**

The specified target does not exist, and there are no commands to execute or inference rules given for it.

U1074 **macro definition too long**

The value of a macro definition would overflow an internal buffer.

U1075 **string too long**

The text string would overflow an internal buffer.

U1076 **name too long**

The macro name, target name, or build-command name would overflow an internal buffer. Macro names may not exceed 128 characters.

U1077 **'*program*' : return code '*value*'**

The given program invoked from NMAKE failed, returning the error code value.

U1078 **constant overflow at '*directive*'**

A constant in the directive's expression was too big.

U1079 **illegal expression : divide by zero present**

An expression tried to divide by zero.

U1080 **operator and/or operand out of place : usage illegal**

The expression incorrectly used an operator or operand. Check the allowed set of operators and their order of precedence.

U1081 **'*program*' : program not found**

NMAKE could not find the given program in order to run it. Make sure that the program is in the current path and has the correct extension.

U1082 '*command*' **: cannot execute command : out of memory**

NMAKE cannot execute the given command because there is not enough memory. Free memory and run NMAKE again.

U1083 **target macro '$(*macroname*)' expands to nothing**

A target was specified as a macro name that has not been defined or has null value. NMAKE cannot process a null target.

U1084 **in-line file not allowed in inference rules**

Inline file syntax is not allowed in inference rules and can be used only in a target-dependency block.

U1085 **cannot mix implicit and explicit rules**

A regular target was specified along with the target for a rule. A rule has the form *.fromext.toext*

U1086 **inference rule cannot have dependents**

Dependents are not allowed when an inference rule is being defined.

U1087 **cannot have : and :: dependents for same target**

A target cannot have both a single-colon (:) and a double-colon (::) dependency.

U1088 **invalid separator '::' on inference rule**

Inference rules can use only a single-colon (:) separator.

U1089 **cannot have build commands for pseudotarget** '*targetname*'

Pseudotargets (for example, .PRECIOUS or .SUFFIXES) cannot have build commands specified.

U1090 **cannot have dependents for pseudotarget** '*targetname*'

The specified pseudotarget (for example, .SILENT or .IGNORE) cannot have a dependent.

U1091 **invalid suffixes in inference rule**

The suffixes being used in the inference rule are not part of the .SUFFIXES list.

U1092 **too many names in rule**

An inference rule cannot have more than one pair of extensions as a target.

U1093 **cannot mix special pseudotargets**

It is illegal to list two or more pseudotargets together.

U2001 **no more file handles (too many files open)**

NMAKE could not find a free file handle. One of the following may be a solution:

- Reduce recursion in the build procedure.
- In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow more open files. FILES=20 is the recommended setting.

U4001 **command file can be invoked only from command line**

A command file cannot be invoked from within another command file. Such an invocation is ignored. The command file should contain the entire remaining command line.

U4002 **resetting value of special macro '*macroname*'**

The value of a macro such as $(MAKE) was changed within a description file. The name by which this program was invoked is not a tagged section in the TOOLS.INI file.

U4003 **no match found for wild card '*filename*'**

There are no filenames that match the specified target or dependent file with the wildcard characters asterisk (*) and question mark (?).

U4004 **too many rules for target '*targetname*'**

Multiple blocks of build commands were specified for a target using single colons (:) as separators.

U4005 **ignoring rule '*rule*' (extension not in .SUFFIXES)**

The rule was ignored because the suffix(es) in the rule are not listed in the .SUFFIXES list.

U4006 **special macro undefined : '*macroname*'**

The special macro name is undefined and expands to nothing.

U4007 **file name '*filename*' too long; truncating to 8.3**

The base name of the file has more than eight characters, or the extension has more than three characters. NMAKE truncates the name to an eight-character base and a three-character extension.

U4008 **removed target '*target*'**

Execution of NMAKE was interrupted while NMAKE was trying to build the given target, and therefore the target was incomplete. Because the target was not specified in the .PRECIOUS list, NMAKE has deleted it.

U4009 **duplicate in-line file '*filename*'**

The given filename is the same as the name of an earlier inline file. Reuse of this name caused the earlier file to be overwritten. This will probably cause unexpected results.

Appendix E
International Character Sort Order Tables

The Setup program used to install BASIC on your computer prompts you to choose one of the four alphabets listed in this appendix for ISAM sorting.

Each alphabet lists primary sort order vertically, and secondary sort order horizontally. Primary sort order is the order in which letters sort, and secondary sort order is the order in which accented characters sort in otherwise identical words.

If you want to change to a different alphabet, you must run Setup again and re-create your indexes.

English, French, German, Italian and Portuguese

```
A,À,Á,Â,Ã,Ä,Å,a,à,á,â,ã,ä,å
B,b
C,Ç,c,ç
D,d
Đ,ð
E,È,É,Ê,Ë,e,è,é,ê,ë
F,f
G,g
H,h
I,Ì,Í,Î,Ï,i,ì,í,î,ï
J,j
K,k
L,l
M,m
N,Ñ,n,ñ
O,Ò,Ó,Ô,Õ,Ö,o,ò,ó,ô,õ,ö
P,p
Q,q
R,r
S,s
T,t
U,Ù,Ú,Û,Ü,u,ù,ú,û,ü
V,v
W,w
X,x
Y,Ÿ,y,ý,ÿ
Z,z
Þ,þ
Ø,ø
```

ß Sorts like and is equal to "ss"
Æ,æ Sort like and are equal to "ae"

Dutch

```
A,À,Á,Â,Ã,Ä,Å,a,à,á,â,ã,ä,å
B,b
C,Ç,c,ç
D,d
Đ,ð
E,È,É,Ê,Ë,e,è,é,ê,ë
F,f
G,g
H,h
I,Ì,Í,Î,Ï,i,ì,í,î,ï
J,j
K,k
L,l
M,m
N,Ñ,n,ñ
O,Ò,Ó,Ô,Õ,Ö,o,ò,ó,ô,õ,ö
P,p
Q,q
R,r
S,s
T,t
U,Ù,Ú,Û,Ü,u,ù,ú,û,ü
V,v
W,w
X,x
Y,Ý,y,ý
Z,z
Þ,þ
Ø,ø
```

```
ß      Sorts like and is equal to "ss"
Æ,æ    Sort like and are equal to "ae"
ÿ      Sorts like and is equal to "ij"
```

Danish, Finnish, Icelandic, Norwegian, and Swedish

```
A,À,Á,Â,Ã,a,à,á,â,ã
B,b
C,Ç,c,ç
D,d
Đ,ð
E,È,É,Ê,Ë,e,è,é,ê,ë
F,f
G,g
H,h
I,Ì,Í,Î,Ï,i,ì,í,î,ï
J,j
K,k
L,l
M,m
N,Ñ,n,ñ
O,Ò,Ó,Ô,Õ,o,ò,ó,ô,õ
P,p
Q,q
R,r
S,s
T,t
U,Ù,Ú,Û,Ü,u,ù,ú,û,ü
V,v,W,w
X,x
Y,Ÿ,y,ý,ÿ
Z,z
Þ,þ
Æ,æ
Ø,ø
Å,å
Ä,ä
Ö,ö
```

ß Sorts like and is equal to "ss"

Spanish

```
A,À,Á,Â,Ã,Ä,Å,a,à,á,â,ã,ä,å
B,b
C,Ç,c,ç
Ch,ch
D,d
Đ,ð
E,È,É,Ê,Ë,e,è,é,ê,ë
F,f
G,g
H,h
I,Ì,Í,Î,Ï,i,ì,í,î,ï
J,j
K,k
L,l
LL,ll
M,m
N,n
Ñ,ñ
O,Ò,Ó,Ô,Õ,Ö,o,ò,ó,ô,õ,ö
P,p
Q,q
R,r
S,s
T,t
U,Ù,Ú,Û,Ü,u,ù,ú,û,ü
V,v
W,w
X,x
Y,Ÿ,y,ý,ÿ
Z,z
Þ,þ
Ø,ø
```

```
ß      Sorts like and is equal to "ss"
Æ,æ    Sorts like and is equal to "ae"
LL,ll  is one character and sorts between "l" and "m"
Ch,ch  is one  character and sorts between "c" and "d"
```

Index

B

D

F

G

J

K

N

O

P

Q

R

U

V

W

MICROSOFT PRODUCT ASSISTANCE REQUEST

Microsoft Product Support Services - Phone (206) 454-2030

Instructions

When you need assistance with a Microsoft product, call our Product Support Services group at (206) 454-2030. So that we can answer your question as quickly as possible, please gather all information that applies to your problem. Note or print out any on-screen messages you get when the problem occurs. Have your manual and product disks close at hand and have all the information requested on this form available when you call.

Diagnosing a Problem

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

1. Can you reproduce the problem?
   ❐ yes ❐ no
2. Does the problem occur with another copy of the original disk of your Microsoft Software?
   ❐ yes ❐ no
3. Does the problem occur with another system (if available)?
   ❐ yes ❐ no
4. If you were running other windowing or memory-resident software at the same time, does the problem also occur when you don't use the other software?
   ❐ yes ❐ no

Product

____

Product name

____________ ____________

Version Number Registration Number

Software

Operating System

____

Name/Version number

Windowing Environment

If you were running Microsoft Windows or another windowing environment, give name and number of windowing software:

____

CD ROM Software

____

Name/Version number

Other Software

Name/Version number of any other software you were running when problem occurred, including memory-resident software (such as keyboard enhancers or print spoolers):

____

____

____

Hardware

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

Computer

____________________ _____________
Manufacturer/model Total memory

Floppy-disk drives
Number: ❐ 1 ❐ 2 ❐ Other
Size: ❐ 3 1/2" ❐ 5 1/4"
Number of Sides: ❐ 1 ❐ 2
Density: ❐ Single ❐ Double ❐ Quad
Capacity:
5 1/4": ❐ 160K ❐ 360K ❐ 1.2 megabytes
3 1/2": ❐ 360K ❐ 400K ❐ 720K ❐ 800K ❐ 1.4 megabytes

System Memory

____________________ _____________
Manufacturer/model Total memory

(If using DOS, you can run CHKDSK to determine the amount of memory available. If using Apple Macintosh Finder, select "About The Finder..." from the Apple menu to determine the amount of memory available.)

Peripherals

Hard Disk

____________________ _____________
Manufacturer/model Capacity(megabyte)

Printer/Plotter

____
Manufacturer/model

❐ Serial ❐ Parallel

Printer peripherals, such as font cartridges, downloadable fonts, sheet feeders:

____

____

Mouse
Microsoft Mouse: ❐ Bus ❐ Serial ❐ InPort™ ❐ Other

____
Manufacturer/model

Boards
❐ Add-on RAM board

____
Manufacturer/model

❐ Graphics-adapter board

____
Manufacturer/model

❐ Other boards installed

____
Manufacturer/model

Modem

____
Manufacturer/model

CD ROM Player

____
Manufacturer/model

Version of Microsoft MS-DOS® CD ROM Extensions:

____

Network

Is your system part of a network? ❐ Yes ❐ No

____
Manufacturer/model

What hardware and software does your network use?

____

____

____

Documentation Feedback: Microsoft BASIC Version 7.0

Help us improve future documentation. When you become familiar with our product, please complete and return this postage-paid mailer. Comments and suggestions become the property of Microsoft Corporation.

Programming experience

_______ Total years _______ Years in this language _______ Months with this product

How do you divide your program development time (%) between the following methods?

_______ % within QBX _______ % using other editor and compiling with BC

Please rate the following documentation features on a scale of 1 (poor) to 5 (excellent):

Poor				*Excellent*		*Comments*
1	2	3	4	5	Installation instructions	_______
1	2	3	4	5	Examples	_______
1	2	3	4	5	Index	_______

How often do you use the following?

Never				*Often*	
1	2	3	4	5	Online Help
1	2	3	4	5	BASIC Language Reference
1	2	3	4	5	Programmer's Guide
1	2	3	4	5	Other (list)

Comments _______________________

Do you find the information you are looking for quickly and easily in the following:

Never				*Often*	
1	2	3	4	5	Online Help
1	2	3	4	5	BASIC Language Reference
1	2	3	4	5	Programmer's Guide
1	2	3	4	5	Other (list)

Comments _______________________

Was it easy to learn the following topics from the documentation?

Difficult				*Easy*	
1	2	3	4	5	Using QBX
1	2	3	4	5	Programming in BASIC
1	2	3	4	5	Mixed-language programming
1	2	3	4	5	Programming with ISAM
1	2	3	4	5	Managing memory use
1	2	3	4	5	Controlling program size
1	2	3	4	5	Using overlays
1	2	3	4	5	Using BASIC toolboxes
1	2	3	4	5	Using command-line tools (BC, LINK, LIB, etc.)

Comments _______________________

Do you want more information on programming:

☐ No ☐ Yes On what topics: _______________________

What do you find the most and least useful about the following documentation items? Why?

Online Help ______________________

____

____

BASIC Language Reference ______________________

____

____

Programmer's Guide ______________________

____

____

If you found errors in the documentation, please give a description: ______________________

____

____

Did we provide all the information you needed? How can we improve our documentation? ______________________

____

____

____

Additional comments: (please attach sheet)

Name ______________________

Address ______________________

City ______________________ State ____________ ZIP ____________

Phone (home) ______________________ (work) ______________________

May we contact you for additional information? ☐ Yes ☐ No

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 108 REDMOND, WA

POSTAGE WILL BE PAID BY ADDRESSEE

Microsoft Corporation
User Education
Data Access Business Unit

One Microsoft Way
Redmond WA 98052-9953